Top K Words Using Map Reduce and Hive

1. Introduction

Google's CEO, Eric Schmidt said: "There were 5 exabytes of information created by the entire world between the dawn of civilization and 2003. Now that same amount is created every two days." Essentially, there is a lot of data. Efficiently storing, processing and analyzing this data is the crux of "Big Data". The extent of the growth of data has motivated the creation of Big Data tools. These tools can be used to successfully store and process huge amounts of data. The two tools we focused on in this project were: Hadoop MapReduce and Apache Hive. There is a lot of data and it's mostly unstructured. So, where do we start? Hadoop helps in filtering and aggregating data to leverage analytics that are important to most businesses today. Hadoop offers various coding options to write programs that can deal with Big Data the right way. The data is distributed across nodes and processed in parallel which is what improves the efficiency. Hive is an open source alternative so that programmers could do the same thing but with fewer lines of code.

In this project, the main objective was to find the top K words in an input file using Hadoop MapReduce and Hive where K is 100 and these are the 100 most frequently occurring words i.e. how many times each word occurs in the text file and then print the top 100 most frequent words. The second part was finding top 100 most frequently occurring words of length greater than 6. One text file is used as input which is of size 32GB. Section 1 introduces the project, section 2 is details of the dataset and tools used, section 3 explains the algorithms used. The graphical results and analysis are showcased in section 4 and section 5 concludes the paper.

2. Experimental Setup

There was one input dataset that was used which was of size 32GB. The 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.

There were two experiments done: 1. Find top 100 most frequent words in the input file. 2. Find top 100 most frequent words of length greater than 6 in the input file. The experiments were first done on Hadoop MapReduce with various different tuning parameters. Secondly, the experiments were executed using a Hive script. All the experiments were run on a Hadoop cluster and this cluster had 23 nodes. The language I used to write the mapper and reducer was Python. Hive scripts were written in SQL.

3. Method

3.1 MapReduce

3.1.1 Case 1: Multiple Reducers

A basic MapReduce program consists of a mapper and a reducer. In this case the mapper parses through the input file, takes in each word and puts a count of 1 next to it and outputs <word, 1>. Then the reducer, for each key which is the word, sums up the count and outputs <word, count>. Then this is sorted in descending order to output the top 100 most frequent words. For, the second experiment, the only difference is in the mapper where there is a condition that the length of each word should be greater than 6. As a first case, I have used multiple reducers and added a MapReduce program to print out the top 100 words. The subcases are described in section 3.1.3.

3.1.2 Case 2: One Reducer

For the above case, since multiple reducers were used an additional MapReduce program was chained to print the top 100 words. In case 2, however, 1 reducer is used which makes possible the reducer sorting and printing the top 100 words. There is no additional MapReduce program chained for this case. 240 mappers are used which are chosen by default based on input and block size and 1 reducer. The subcases are described in section 3.1.3.

3.1.3 Subcases

a. Mapper & Reducer

The first case used a mapper and reducer to count the frequency of the words and output the top 100 words in the text file. It uses 240 mappers which are chosen by default based on input and block size. Case 1 used 96 reducers and case 2 used 1 reducer.

b. Mapper, Reducer & Combiner

In MapReduce, an intermediary program can be used called a combiner which would potentially improve performance. In our case, since the keys don't change the reducer script can be used for the combiner as well. Two experiments are done: one for all words, one for words of length greater than 6. 240 mappers are used which are chosen by default based on input and block size. Case 1 used 96 reducers and case 2 used 1 reducer.

c. Mapper, Reducer, Combiner with Partitioner

A partitioner can be used to control the partitioning of the keys in the intermediate mapper output. The total number of partitions depends on the number of reduce tasks. This improves performance as records with same key value go into the same partition. In case 3, partitioner is used to sort the words and count into appropriate partitions so it improves the performance in the combiner and reducer steps. 240 mappers are used which are chosen by default based on input and block size. Case 1 used 96 reducers and case 2 used 1 reducer.

d. Mapper, Reducer & Combiner using Compression of Text File

Our input file is huge: 32GB; most big data files would be. Therefore, compression techniques before and during map reduce programs become necessary. In case 4, the MapReduce output was compressed until it went to reducer where it was automatically decompressed. This was done for

both experiments. 240 mappers are used which are chosen by default based on input and block sizes. Case 1 used 96 reducers and case 2 used 1 reducer.

3.1.4 Case 3: Varying the number of Reducers

Since the number of reducers is an important parameter for fine tuning a MapReduce program, case 3 tried two different values for the number of reducers used: 30, 60. These are in addition to the cases using 1 reducer and 96 reducers. The number of mappers remained 240. A combiner was used in this case as well.

3.2 Hive

3.2.1 Case 1

Hive uses SQL script where you can create a table, add your data to that table then make queries. Each query would be a MapReduce program in the previous cases and therefore Hive is easier to learn and code. I created a table for the input file and a table for the words. Using a python script, the lines were split, and words were added to the table. For the first experiment I used count for each word to find the count and used a limit of 100 along with group by words and order by count in descending order. For the second experiment I used count again but this time there was an extra condition of length of word had to be greater than 6. All of this was added to one hive script which can be run in its entirety.

3.2.2 Case 2: Indexing

Indexing for a Hive script can be used to improve efficiency. When you create an index, it acts as a reference to the records for the table you created the index for. The index could be used to search for a particular record making searching easier. Indexing was used for the table with words from the text file before performing querying on the table.

3.2.3 Cases 3 and 4: Varying Mappers and Reducers

The hive script was using mappers and reducers to run its queries. I tried to vary the number of mappers and reducers to find the best combination. Three combinations were tried: [250, 600], [150, 500] and [115, 468].

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.

4.1 MapReduce

Figures 1, 2 and 3 showcase the execution times for the two cases for experiment 1 and for the first case for experiment 2. As can be seen in the results, the performance (execution time) for all three graphs was lowest (best) for Case B, which is using a mapper, reducer with a combiner. Using a combiner significantly improves efficiency. On the contrary, for this problem using a partitioner or compression techniques didn't improve the execution time.

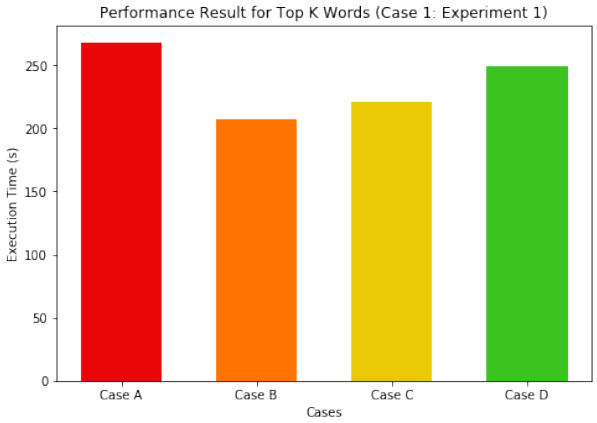

Figure 1: Performance of MapReduce for Case 1: Experiment 1

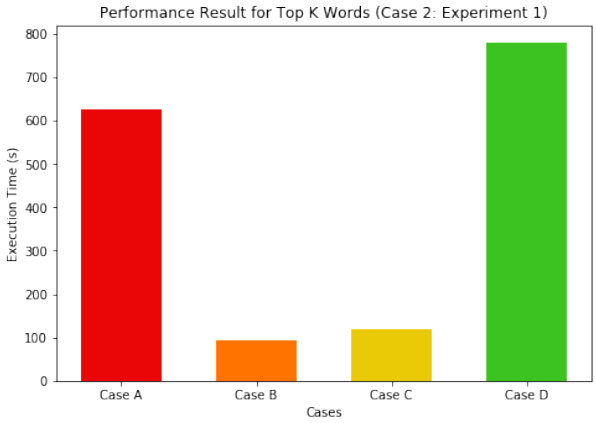

Figure 2: Performance of MapReduce for Case 2: Experiment 1

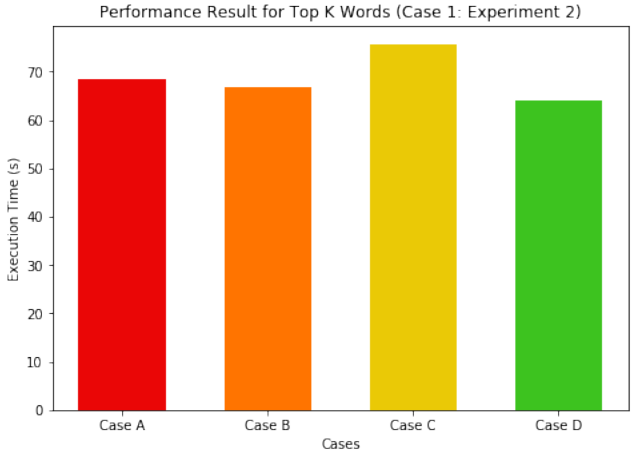

Figure 3: Performance of MapReduce for Case 1: Experiment 2

Case 1: Using Multiple Reducers (96) with 240 Mappers.
Case 2: Using 1 Reducer with 240 Mappers.

Case A: Mapper and Reducer
Case B: Mapper, Reducer and Combiner
Case C: Mapper, Reducer, Combiner and Partitioner
Case D: Mapper, Reducer, Combiner and Compression of Mapper Output

Case 3 varied the number of reducers including the number of reducers used in cases 1 and 2. In addition to 1 and 96 reducers used in cases 1 and 2, 30 and 60 reducers were used for each experiment. For both experiments, 1 reducer executed in the least time as can be seen in figures 4 and 5. Case 3 was run using Case B (mapper, reducer and combiner) as that was the best case in terms of execution time from previous analysis.

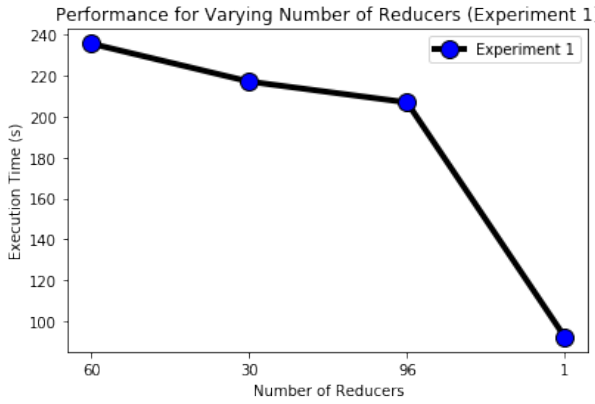

Figure 4: Performance for varying number of reducers for Case B for Experiment 1 (MapReduce)

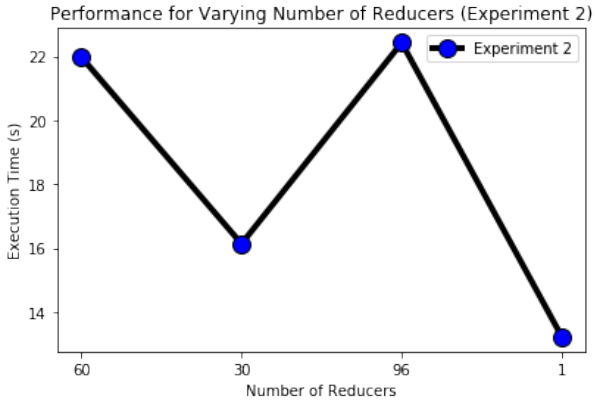

Figure 5: Performance for varying number of reducers for Case B for Experiment 2 (MapReduce)

4.2 Hive

The first Hive script that I run was a simple script where it created a table for the words from the input file and each experiment was a query; this was case 1. Case 2 then used indexing to improve efficiency. The improvement is showcased in figure 6. This is for experiment 1.

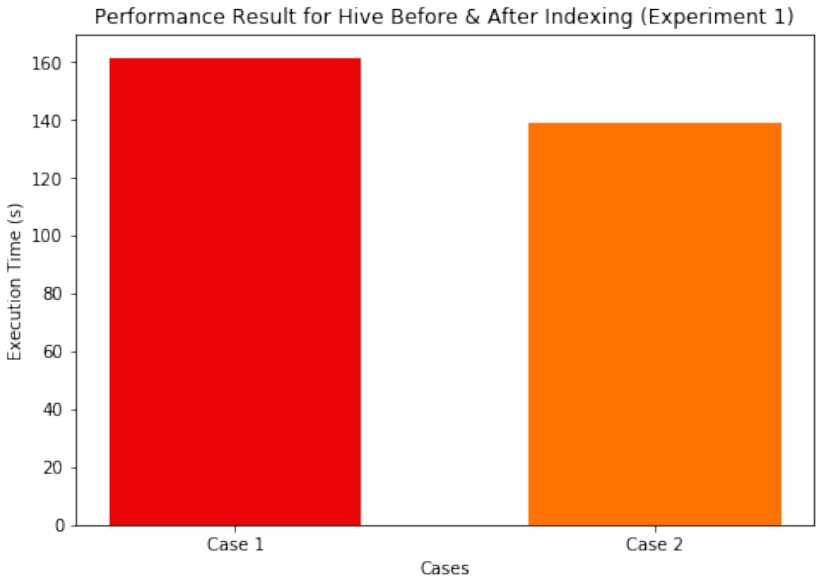

Figure 6: Performance before and after indexing (Hive)

For experiment 2, after using indexing, I further tried to optimize by trying to vary the number of mappers and reducers used as Hive queries and jobs used mappers and reducers to execute. Figure 7 showcases these results; the best execution time was for 115 mappers and 468 reducers.

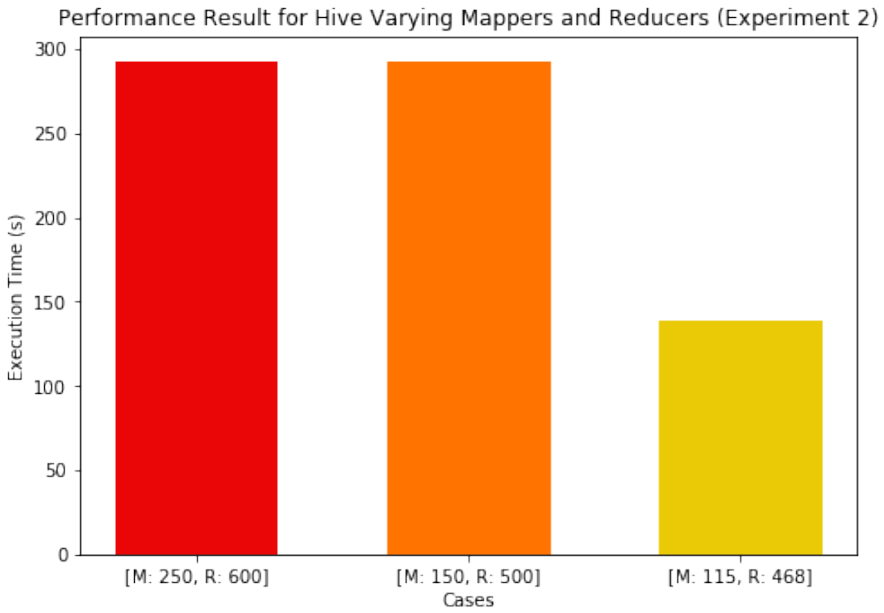

Figure 7: Performance difference varying number of reducers. (Hive)

4.3 MapReduce VS Hive

MapReduce and Hive both worked efficiently for this problem when compared to the execution of a solution that doesn't use a Big Data Tool i.e. a simple python program. However, between MapReduce and Hive, which is the better tool depends on the problem. In Table 1, I've written some of my observations that I made while using both tools throughout the project.

	MapReduce	**Hive**
Number of Lines of Code	More	Less
Ease of Use	Easier in terms of parameter tuning as there is more well documented examples available and running the programs repeatedly is easier.	Easier in terms of writing the code as SQL has lesser lines of code and it takes lesser time to learn and be able to write the program.

Table 1: MapReduce VS Hive

In terms of performance for this project, figure 8 showcases the difference in execution times for MapReduce and Hive for both experiments. Hive does significantly better for experiment 1 where there is no condition in place. However, for experiment 2 with a condition, MapReduce's performance is much better. We could infer that picking the best Big Data tool to use would heavily depend on the specific problem at hand and the dataset. In this case when an extra condition is in place (length of word has to be greater than 6) MapReduce works better. But for experiment 1 where it's simply printing top 100 words Hive query works faster.

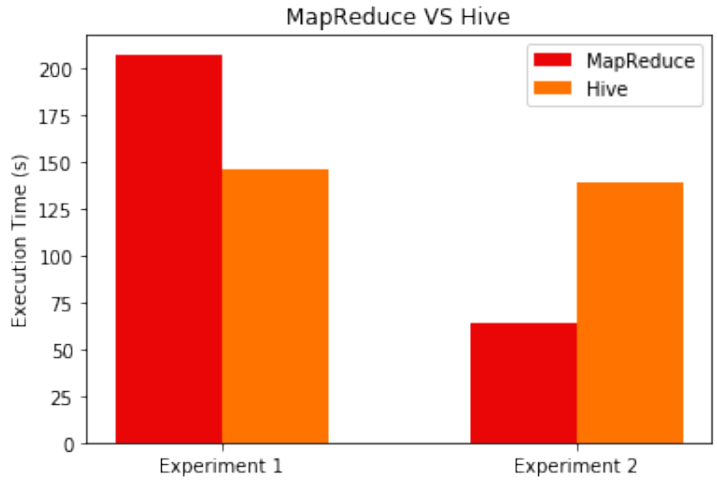

Figure 8: Performance of MapReduce versus Hive for both experiments

5. Conclusion

The crux of Big Data is being able to efficiently read and process large files; the main objective of this project was to use two popular Big Data tools: MapReduce and Hive and analyze their performance. This project showcased some of the techniques that can be used to read and process a large dataset (in our case, a 32GB file). In the first project, the best performance I'd gotten for this file size is much worse than MapReduce and Hive. Figure 9 clearly showcases the advantage of using these Big Data tools for large datasets in terms of execution time where the python program is the best execution time from the first project. Big Data tools provide a framework for efficient processing of large datasets; a solution that is essential today. However, deciding which tool to use depends heavily on the specific problem we're dealing with, the dataset contents and the size of the dataset. Overall, both MapReduce and Hive solved the problem in this project more efficiently than a simple python program using several nodes and mapper/ reducer tasks proving the importance of Big Data tools today.

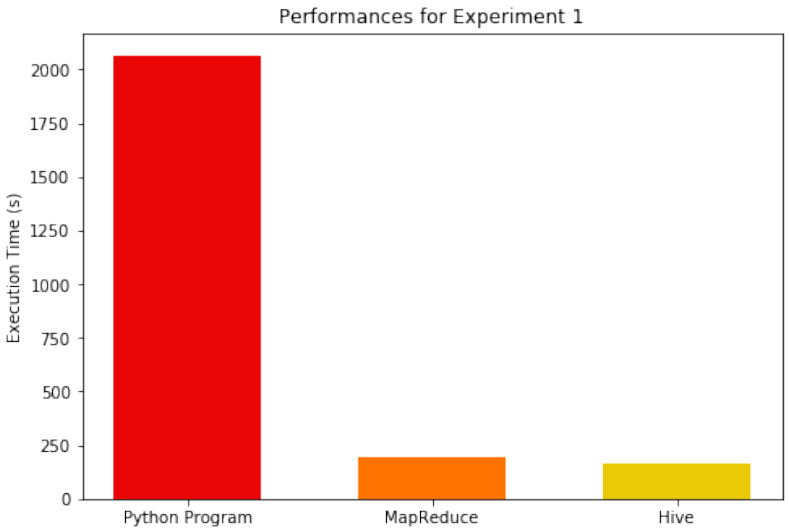

Figure 9: Performance of different techniques for same problem.

References

1. Pusukuri, Kishore. COEN 242 Class Slides. Santa Clara University, 2020
2. "Hive Useful Resources" https://www.tutorialspoint.com/hive/hive_useful_resources.htm
3. "MapReduce VS Pig VS Hive" https://www.dezyre.com/article/mapreduce-vs-pig-vs-hive/163
4. "Hadoop Streaming" https://hadoop.apache.org/docs/r1.2.1/streaming.html#Generic+Command+Options
5. "Python Word Count" https://cwiki.apache.org/confluence/display/HADOOP2/PythonWordCount
6. "Hadoop Python MapReduce Tutorial for Beginners" https://blog.matthewrathbone.com/2013/11/17/python-map-reduce-on-hadoop-a-beginners-tutorial.html