Table of Contents

Overview of the assignment

We are provided with a text file containing 10,000 amazon reviews and a list of keywords and excluded words. The reviews are divided into a title, review, and sentiment of 1 being a negative review, and 2 representing a positive review. Our task in this assignment is to retrieve the reviews which have the keyword present in them, exclude the excluded words, and fetch the most appeared word along with their count, and the total word count present in the reviews.

Making the directories

The first thing we need to do is to log in to the university's Hadoop server using student username. We then create the relevant directories for the assignment using the following commands.

mkdir ITNPBD7/hadoop/assignment

To change the directory from home directory to the directory we made for the assignment, the command we use is

cd ITNPBD7/hadoop/assignment

In the above directory, we upload the files provided for the assignment locally. Following are the files that we uploaded.

- Reviews.txt, 5.txt, 100.txt
- Mapper.py, combiner.py, reducer.py
- Runhadoop.sh, simhadoop.sh

HDFS

To run our reviews sentiment analysis on Hadoop Distributed File System, we need to understand how the HDFS works. The HDFS works best on huge amount of data, so let us assume that our reviews.txt file is much larger than it is.

- As we send our reviews file across the HDFS, it gets divided in 64MB blocks or chunks, with each chunk containing the different title, review and the sentiment related to it. The data stored in these blocks is randomly stored.
- These blocks are then sent across different data nodes across the cluster. HDFS by default will make a replication of the file three times and store in different data nodes so that if one of the data nodes fail, we will have the replicas across other data nodes.
- Each data node will have a mapper running in it and the computation that is done is parallel. So, when each of our mapper runs inside each of our data node parallelly, we will emit key value pairs for our reviews.txt file. The key being the keyword and their sentiment (e.g., cat 1, dog 2) and the value being a dictionary of the word and their counts in each review.
- The output of the mapper will then become the input of our reducer, which will then make a reducer for each distinct key that the mapper emits. So, for our review's sentiment analysis, the reducer will make ten different keys, one for each keyword and their associated sentiment. The

reducer will then calculate which total word counts and the most common word and their counts for each distinct key.

The commands that we use for sending the data to the HDFS and run our code are as follows:

- HDFS makes the files permissions set to read and write only by default. To make sure that our runhadoop.sh, mapper.py, reducer.py, and combiner.py files are executable too, we use the following commands in the terminal

  chmod u+rwx *.py

  chmod u=rwx runhadoop.sh

- To make a directory for the sentiment analysis in the HDFS itself, we use the following command.

  hdfs dfs -mkdir sent_analysis

- Now, we need to copy our reviews.txt file from local system to the sent_analysis directory in the HDFS

  hdfs dfs -copyFromLocal reviews.txt sent_analysis

- To check if our reviews.txt has been transferred to HDFS, we use '-ls' command which shows us that the reviews.txt file has been sent to sent_analysis directory

  hdfs dfs -ls sent_analysis

- Now that our reviews file has been sent to the HDFS to process, we can run our hadoop job by executing the runhadoop.sh file. The runhadoop.sh tell hadoop to run each file in a certain order. The HDFS will take input from reviews.txt to the mapper.py file, after the parallel processing in the mapper, the output of the mapper is sent as an input to either the combiner or the reducer. The following command runs the job:

  ./runhadoop.sh

- After running the job successfully, the results will be placed in the following directory inside the HDFS:

  INFO streaming.StreamJob: Output directory: /user/msz/sent_analysis/results

- *To see our results in the terminal, we can use the following command:*

  hdfs dfs -cat sent_analysis/results/part-00000

- *Now, if we want to save our output results in our local file store from HDFS, we can use the -copyToLocal command as follows*

  hdfs dfs -copyToLocal sent_analysis/results/part-00000 results.txt

- *For testing purposes, we can use concatenate and piping commands which will show the results on standard output or terminal*

  cat reviews.txt | ./mapper.py | ./combiner.py | sort | ./reducer.py

Design

Mapper stage

After the standard input from reviews.txt is distributed in various 64MB blocks, these blocks are sent across to different data nodes and replicated three times by defaults, in case if a data nodes crashes down. In this assignment, we are dealing with 10,000 reviews, but let us assume out data set is much bigger, and it contains 1 million reviews, when we send these reviews across thousands of different data nodes, the task is divided among each data node and mapper will work on it parallelly for efficiency.

In our mapper.py file, we have different processes that we are going to do on our reviews.txt file

- We start off by defining our 'removePunctuation' function, which will remove all the unnecessary words from the reviews. I chose to not use the function already provided on the mapper file because when I got the output of my mapper and reducer using simhadoop.sh, there were still some special characters that were appearing in the reviews (e.g., */+). Therefore, I decided to replace it with "Regular Expression (RegEx)", which keeps all the characters that start from A-Z, a-z, 0-9, and removes every other special character that appears in the reviews to make my word frequency calculations accurate later.
- After fetching all the reviews with keywords (cat, dog, movie, music, and phone) in it and excluding all the words in the excluded.txt file, I used a regex function to find the frequency of each word that appeared in the review. The aim to find one layer of word count for each review was to minimize the workload of the reducer.
- The mapper takes each line from standard input and emits a key-value pair. The **key** is the keywords and their sentiment (e.g., cat 1, dog 1), and the **value** being the dictionary of the frequency of word counts. The dictionary has words as *key*, and their respective count as the *value*. I opted to go for this output structure because out of 10,000 given reviews, we have extracted 2,260 reviews after running our mapper code. Instead of emitting the key (keyword and rating) and value(one word and its respective count in each review) in one line, the dictionary here is more efficient because it will cause the reducer or combiner to go through less lines of data.
- To have an idea, the following image (Figure 1) shows a part of my mapper output.

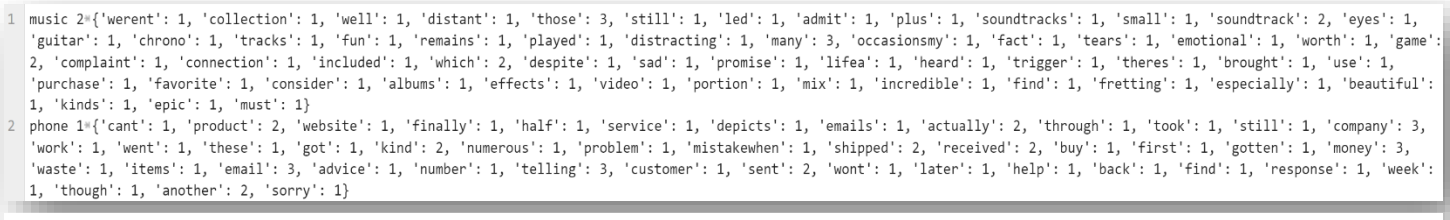

Figure 1: Mapper output

Combiner stage

The combiner works as a mini reducer to reduce the workload of the reducer. When we will run our ./runhadoop.sh job in the HDFS, the combiner may or may not run because the reducer does not actually know that the output is coming from mapper or a combiner. Therefore, our combiner output structure should always be the same as the mapper output structure. I constructed my combiner code in such a way that the reducer code did not need to be changed or improvised.

As we know that we have our reviews.txt file distributed over many data nodes and the mapper works on them parallelly, the main logic behind my combiner code is to minimize the data that is going to the reducer, by concatenating the word counts of the reviews with keys that come up in each data node consecutively. By using this logic in my combiner, I was able to decrease the mapper output from **2,260** reviews to **1400** reviews/lines in the output of the combiner. As fewer data moves through the network to the reducer from the combiner after concatenating the consecutively placed reviews in data nodes, this improves the efficiency of the overall design.

One important thing I had to deal with while making my combiner and reducer code was that the mapper output, I assumed was emitting a dictionary as values, but across the network, the data travels as a string, so I had to convert the string into a dictionary by importing "ast" library from python.

To have a better understanding how the combiner is working, the following image (Figure 2) shows the idea behind the combiner.

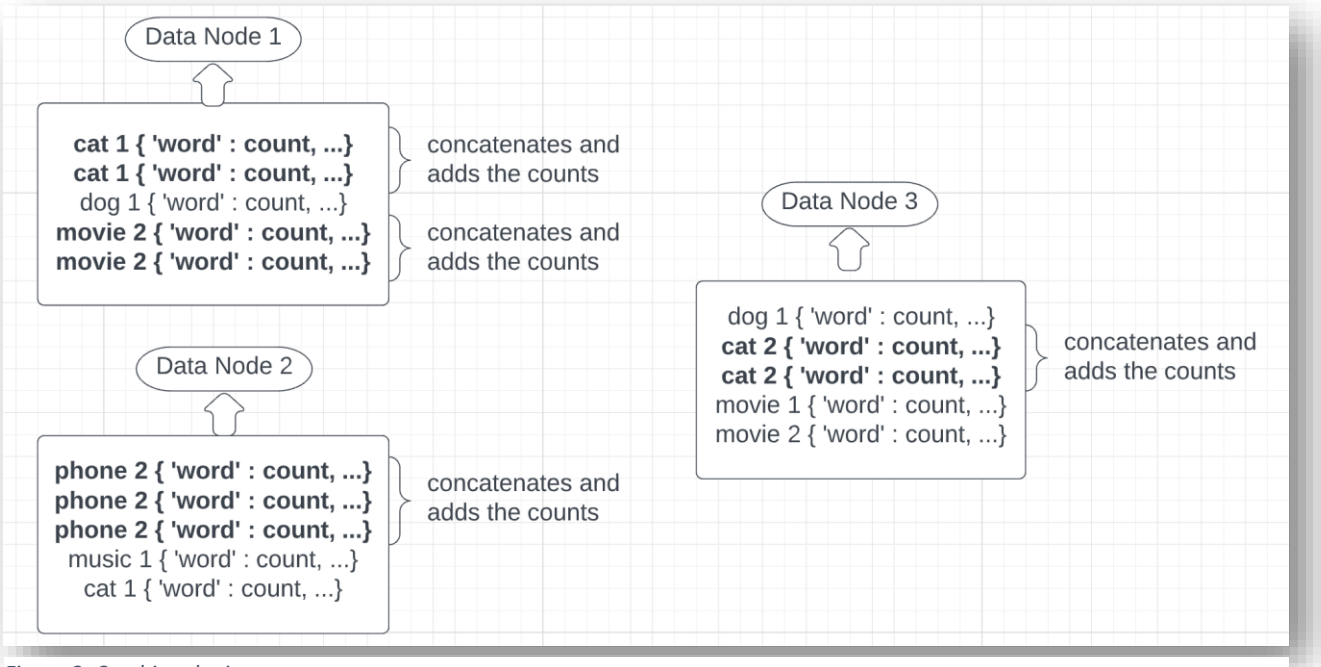

Figure 2: Combiner logic

Reducer stage

The reducer takes the input from either the mapper or combiner. The reducer in our review's sentiment analysis, will calculate the total word count, and the count of the most common word for each distinct key.

- Before we move on the reducer, we sort the output of the mapper/combiner so that all the keys are in alphabetical order.
- From the standard input, there will be a reducer made for each distinct key. E.g., The first key that will come through from the standard input after sorting the mapper/combiner will be "cat 1", so the reducer will make a separate reducer for "cat 1". Then, when the key changes to "cat 2", another reducer will be made, and so on for other distinct keys. In total, for our assignment, we will have 10 reducers.
- In our reducer code, we first convert our mapper/combiner output into a dictionary using "ast". Then, we run an "if" loop checking if the current key is equal to the review key and increment the count to the common words in each key. When the key changes, we change our current key and do the process all over again.
- In my code, I went for an unconventional way and made an "outer_dict" which stores 10 distinct **keys** for each keyword and rating, and **values** as the overall review's frequency (words and their counts). Then, using a 'for loop' in the "outer_dict", I used the function "getMax" which fetches the most appeared word for each key and its count. The total word count is also calculated using the outer_dict.

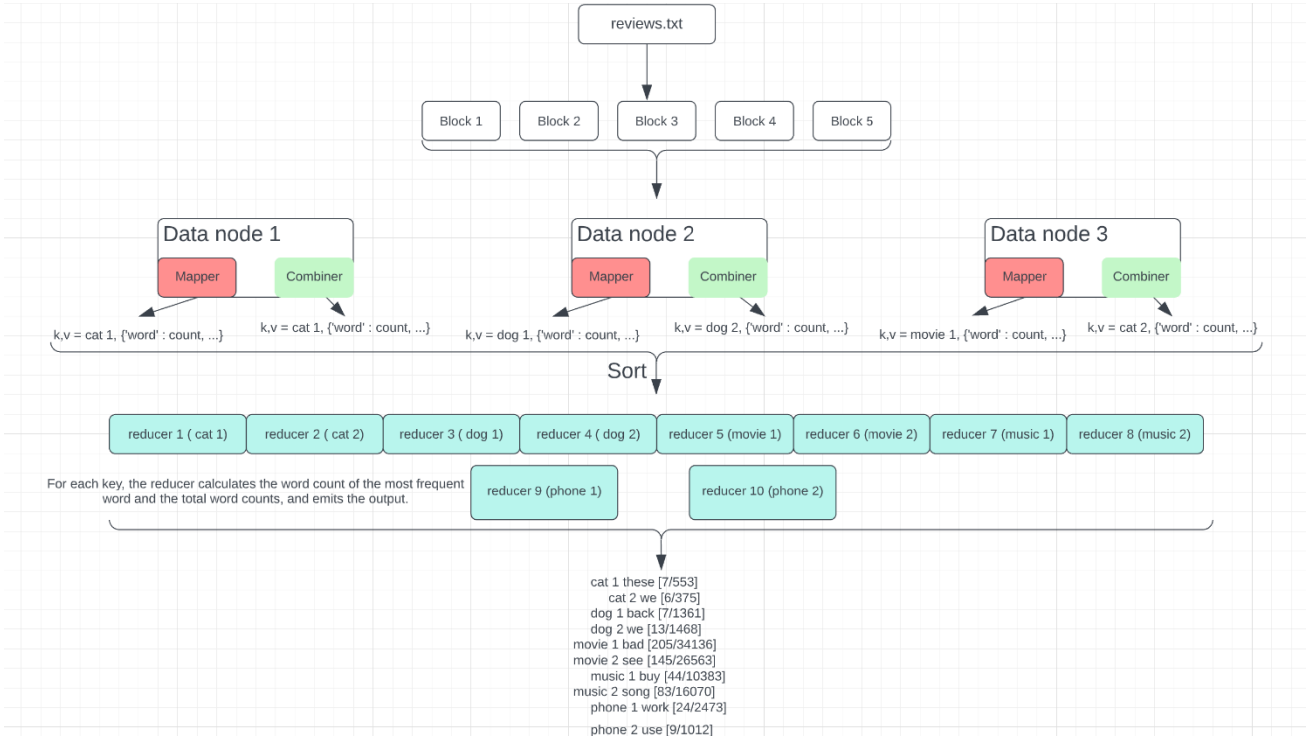

Figure 3: Design of the MR job

Comparison

- When we run the job on only the mapper and reducer, the mapper output will send more data across the network. Although most of the processing is being done in the mapper, the reducer has to concatenate and process 2,260 reviews from the mapper output.
- When the combiner is run along with the mapper and reducer, the combiner lessens some of the processing that the reducer has to do by combining the counts of consecutive keys that appear in the data node. Therefore, the reducer processes 1,400 reviews from combiner output.
- The more efficient solution would be when the map reduce job is run with the combiner. In my observation, the processing time did not improve much with the combiner, but if we assume a much larger data set, the processing time difference would be clearer.