Cloud Computing Coursework Report

Google Storage Paths

Text files for testing:

Gs://cc_coursework_output/input

Output

Gs://cc_coursework_output

Gs://cc_coursework_output/output (for result of processing 10000 books)

Development of program and challenges faced

Mapper

The first challenge during the development of the program was deciding how finding anagrams would be implemented as a MapReduce program. While the actual algorithm to find anagrams would be simple, MapReduce splits the data and processes it in parallel, so a method would be needed to sort all anagrams into the same reducer.

The mapper emits a key-value pair, where the key is used to determine which reducer the output will go to. By sorting each word alphabetically, a key can be created for each word and anagrams will share the same key, grouping them into the same reducer.

Example:

Input 1: Race Input 2: Care

Key: Acer Key: Acer

Output 1: Acer, Race Output 2: Acer, Care

Now that anagrams can be identified the mapper only needs to remove numbers and punctuation as well as ignore stop words. A string variable containing all the given stop words was initialised and split into an array, where the mapper would check if the array contains the word being mapped. If it is not it then it had punctuation and numbers removed, as well as being changed to all lower case. One issue that occurred was that the program would consider a single letter as a word, so the mapper was changed to only accept words with a length above 1.

Example:

Input: Race21!

Output: race

Reducer

Now that the mapper was written the next part to implement was the reducer. Given that the key from the input could be used to identify anagrams all that was needed was to add each word that had the same key to a string that would be outputted by the reducer. As the input from the reducer contained duplicates of words they would first need to be ignored. To do this a list of unique anagrams would need to be made before adding all the words in the list to the output.

Initially an array list was used and a contains() method was used to check if the word was already in the list. Only unique words would be added to the list and the list could then be sorted alphabetically. After that, the list could be iterated through, which would result in each set of anagrams being sorted alphabetically. The output of the reducer was set to show the number of anagrams in the set as well as the set itself

Example:

Input: Care, Race, Care

Set: Care, Race

Output: 2 Care, Race

A major problem that came up was that this did not seem to remove duplicates as the output of the code when testing resulted in each set being a large number of a single word, which indicated that the duplicates were not being removed and for some reason only a single word was being added to the list.

Example:

Output: 150 Race, Race, Race, Race, Race, Race,….

To try to solve this issue the array list was changed to a hash set, which only allowed unique elements to be added to it. This meant that code to check if the word was already in the list was not needed, which removed a possible error in coding. To further optimise the code the hash set was changed to a tree set, which both only allowed unique elements and automatically ordered the elements, so the sort function was also removed.

Array List	Unordered, allows any elements
Hash Set	Unordered, only allows unique elements
Tree Set	Ordered, only allows unique elements

This solution cut down the number of duplicates, but the output of each set still consisted of duplicates of a single word. The number of words in each set had gone down drastically, which showed that the duplicates had been removed. This meant that the words in the set of anagrams were being changed when the list was sorted.

```
3       owns,owns,owns
2       town,town
3       now,now,now
2       posts,posts
5       spot,spot,spot,spot,spot
2       top,top
2       ours,ours
2       out,out
3       two,two,two
alexvong14@cc-coursework:~/hadoop-3.3.0$
```

After some investigation it seemed that when the words in the set were sorted the words themselves were being affected.

Example:

Set: Race, Care

Sorted set: Race, Race

This seemed to stem from the fact that the data type used was Text, not String. While both data types are used to store character values it seems that Text data interacts differently with the sort function. I am unsure as to exactly how the interaction works but changing the input word to a String before adding it to the set resolved the issue.

```
2       posts, spots,
5       post, pots, spot, stop, tops,
2       pot, top,
2       ours, sour,
2       out, tou,
3       tow, two, wot,
alexvong14@cc-coursework:~/hadoop-3.3.0$
```

Optimising the execution time of the program

To reduce the execution time of the program a combiner was implemented in the code. The mapper produces large amounts of data to be sent to the reducer, which results in a lot of network congestion. This can be reduced by using a combiner to process some of the data before it is sent to the reducer, which reduces network congestion and the amount of data the reducer needs to process, which both reduce execution time.

In this case the combiner was used to remove duplicate words from the mapper output. It would attempt to add each word to a hash set and if it were successful (the word was unique) then it would pass that word to the reducer. As combiners are run locally on each mapping machine it will not remove all duplicates, but it does reduce the number of duplicates going into the reducer.

To further reduce execution time the use of speculative execution could be implemented. This would execute slow running map tasks on another node, which should prevent a (few) slow tasks from delaying the reduce execution, as the reducer can only run when all the mapping tasks are completed. It is also possible to tune the performance of the machines by balancing out the reducer loads. This could be done by implementing an improved hash function in the partitioner, which would split the data input evenly between the reducers. This would mean that a reducer would not receive more data than the others and end up running longer.