Big Data Analytics

HomeWork-3

Anjani Swetha Settipalli

bigd24

Question 1:

1.1 Problem Description:

We have a dataset of multiple thousands of books as raw text files from www.gutenberg.org. Given the search term our goal is to identify the most relevant books from the given dataset. In the given problem we build an InvertedIndex using Map reduce to build a simple search engine.

1.2 Solution Strategy:

In order to build an Inverted Index we use two mappers and two reducers.

The inverted Index is an Index data structure storing a mapping from words to its file locations in the database. Inverted Index allows fast full text searches.

Mapper 1:

- The first mapper is basically used to read the given files and count the frequency of words in each file. The following are the steps performed in order to get the word frequency per given document.
- The mapper receives the given database.
- Map reduce job operates on per line basis. We detect the name of the input file using File Split and get the path of the file name.
- All the special characters and the numerical values are removed from the given file. Only alphabets are considered.
- We tokenize the lines to get an individual strings. Check if the given word is in the filler list. If not,
- Set the word and filename together as key and 1 as value similar to the word count.

Reducer1 :

- Reducer combines all the values of a particular key together.
- The reducer adds up the frequency and give frequency of word per particular document as output.
- The key of first reducer is word and file name and the value is the total frequency.

Mapper 2:

Analysis:
In order to get the sorted output of files and the frequencies of the word in a particular file,
There are two possible approaches here. The first approach involves having the reducer buffer all of the values for a given key and do an in-reducer sort on the values. Since the reducer will be receiving all values for a given key, this approach could possibly cause the reducer to run out of memory. The second

approach involves creating a composite key by adding a part of, or the entire value to the natural key to achieve your sorting objectives

The trade off between these two approaches is doing an explicit sort on values in the reducer would most likely be faster, however there is a risk of running out of memory, but implementing a “value to key” conversion approach, is offloading the sorting the MapReduce framework, which lies at the heart of what Hadoop/MapReduce is designed to do . Hence I used the functionality of map reduce framework which automatically sorts the keys.In order to achieve that we use the concept of composite key.

Explanation:

- This mapper gets the token, document Id and frequency as inputs which are the outputs given by the first mapper.
- The composite key is take as token and the frequency. As we need the sorting by frequency we included the token frequency as the part of composite key.
- The values of the mapper is given as doc id and frequency.

Creating a composite key:

In order to create a composite key we need to analyze the what parts of the value we want to account for during the sort and add the appropriate parts to the natural key.

There are four things to be created

1. Map output key class (CompositeKey Class)
2. Partitioner class (ActualKeyPartitioner)
3. Grouping comparator class (ActualKeyGroupingComparator)
4. Comparator class (CompositeComparator)

CompositeKey class:

- CompositeKey class is mostly like a customized writable class where we write the getter and setter classes of the writable.

Partitioner:

- Next step is to implement the **partitioner**. The reason we are implementing our own partitioner is that the default partitioner would not be able to ensure that all the records corresponding to certain word comes to the same reducer(partition).
- Hence we take the partitioner only to consider the actual key part (word) while deciding on the partition for the record.

Grouping Comparator:

- The partitionar only makes sure that the all the records related to the same word comes to a

perticular reducer. But it doesnot ensure that all of them come in the same input group., i.e., in the single reduce call as the list of values. In order to achieve this we use grouping comparator.
- We look only for the actual key i.e., word for grouping the reducer inputs.

Secondary sorting:

- Now we do the secondary sorting over datetime field. In order to achieve this we create our own comparator which check for the equalities of words , if they are equal then we check for the the frequency field.
- Secondary sort is a technique that allows the MapReduce programmer to control the order that the values show up within a reduce function call.

Reducer 2:

- The reducer gets the keys and values and write them to the instance for MultipleOutputs is created.
- Finally the words are grouped into files alphabetically and each file contains words of perticular alphabet

Driver method:

- The driver method reads the filler word file and add them to the array list which is used to remove the filler words from the given program.
- It sets the job of first and the second map reduce programs and also sets the input output key formats.

1.3 Improvements implemented:

Removing quotes from the words:

The quotes special character and the numerical values are removed from the input files
We used the regular expression val.toString().replaceAll("[^a-zA-Z]+"," ");
to achieve this.

Removing the filler words:

The filler words are like (the, a, an, has, if, and, or, I, you,
we, that, they, them, …). In order to remove the filler words all the words that are considered as filler words are saved in a text file.
The text file is read in the driver method and are stored in the array list. While parsing through the key value pairs in the mapper we check the key if it is present in the array list. If it is present it is considered as a filler word and is discarded while sending to the reducer.

Sorting output e.g. into separate files based on the starting character of the term to speed up:

We implement the MultipleOutputs in order to achieve this. We generate the file names based on the starting character of the key i.e., word and then save in the file generated by the starting character of the keys. This helps in great extent in retrieving the values as only those files that are required are processed. This reduces the size of data set tremendously and the processing time decrease to grea extent

1.4 Description of resources used:

Description of Resources Used:

20 SUN X2100 nodes (shark01 - shark20)

- 2.2 GHz dual core AMD Opteron processor
- 2 GB main memory

3 SUN X2200 nodes (shark25 - shark28)

- two 2.2 GHz quad core AMD Opteron processor (8 cores total)
- 8 GB main memory

Network Interconnect

- 96 port 4xInfiniBand SDR switch (gift from Cisco Systems)
- 48 port Linksys GE switch

Storage

- 20 TB Sun StorageTek 6140 array (/home shared with crill)
- 4 TB distributed PVFS2 storage (/pvfs2)

1.5 Result:

Input:
run the following command

```
time yarn jar InvertedIndex.jar InvertedIndex /bigdata-hw3 /bigd24/output/InvertedOutput1/
/bigd24/output/InvertedOutput2
```

Output:

The output comes in different files starting from a to z in the InvertedOutput2 directory.

The output files are of the form a-WORD-r-00000, b-WORD-000000 and so on.

1.6 Measurements:

The performance of the given algorithm is measured by running the program for multiple times and considering the least time taken.

First run:

```
real    13m2.862s
user    0m17.952s
sys     0m1.600s
```

Second run:

```
real    14m2.462s
user    0m18.942s
sys     0m2.651s
```

Third run

```
real    12m2.732s
user    0m17.952s
sys     0m1.500s
```

Fourth run:

```
real    14m1.123s
user    0m1.999s
sys     0m1.600s
```

Fifth run:

```
real    13m2.862s
user    0m17.456s
sys     0m1.674s
```

From the above timings given by the output we consider the user time to compare the performance f the given algorithm

Number of runs	Time taken
1	13m2.862s
2	14m2.462s
3	12m2.732s
4	14m1.123s
5	13m2.862s

From the above timings we can observe that there is some difference in the execution of the algorithm for different runs. This could be due to network traffic. Hence we consider the minimum time of execution as the measure of performance of the given algorithm.

From the above table we can observe that the minimum time taken for the execution of the algorithm is 12m2.732s

2. Performance measure by changing the number of reducers:

With the increase in the number of reducers the load on the single node decreases as the load distributes to different nodes as number of reducers and hence we measure the performance by varying the number of reducers

Command used:

First run: 1 reducer

```
real	18m2.862s
user	0m17.952s
sys	0m1.600s
```

Second run: 5 reducers

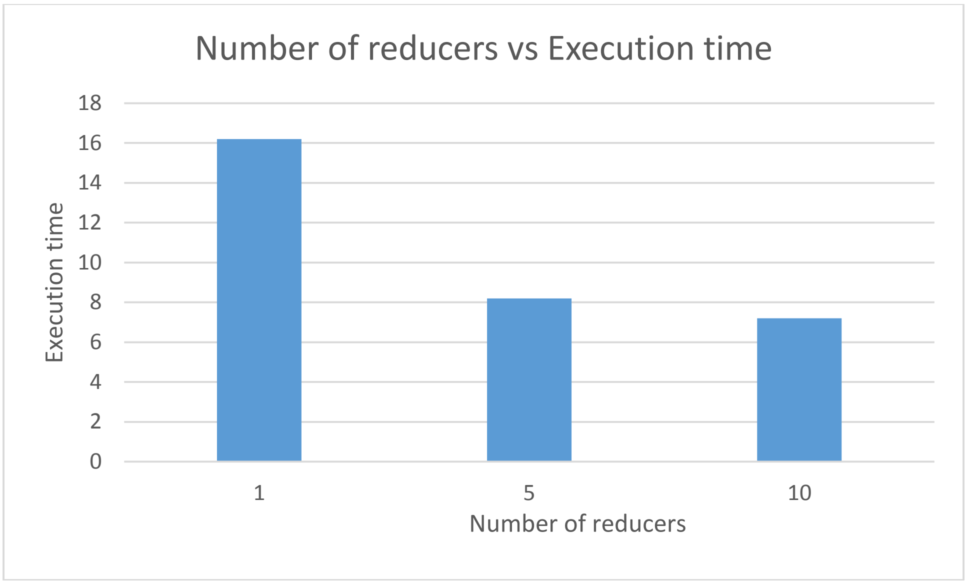
```
real	8m2.462s
user	0m18.942s
sys	0m2.651s
```

Third run 10

```
real	7m2.732s
user	0m17.952s
sys	0m1.500s
```

Number of reducers	Execution time
1	16m2.862s
5	8m2.462s
10	7m2.732s

From the above details we can observe that as the number of reducers increases the execution time decreases. This is because the load is distributed among different nodes as the keys are distributed among different reducers.



From the above plot we can observe that as the number of reducers increases the time required for execution decreases. This is due to distribution of load on different nodes.

Question 2:

2.1 Problem Description:

In the given problem we need to build the retriever from the inverted index built in the first problem . More than one query term is given as inputs . For a given document sum of term frequencies are calculated and the top 10 files that contains high frequencie sof the given words are given as output.

2.2 Solution Strategy:

- From the inverter index multiple files generated from the first problem we build a simple search engine.
- The search engine is implemented through java language.
- Prompt user to enter the words to search .
- When the user enters the words add the words and first letters of the words to an array list
- Iterate through the first letters of the words get the appropriate Inverted Index file which is stored previously with the starting letters of the keys.
- After getting the appropriate files add them to the hash map with word as key and their filename:frequency as the values

- Now sum the frequencies of per documents of the given search words and put them into hash map.

- Finally sort the value by passing the hash map through the TreeMap and display top ten search values for the given search words.

2.3 Results:

Input:

```
Java Search.java
Java java -cp . Search
```

2.4 Resources Used:
Please refer to Section 1.4

2.5 Measurements:

The execution time for the search of search of different terms are compared.

When searched for a single word

```
Search for words: blue
wldfl10.txt:219
whewk12.txt:142
cusa110.txt:134
shlyc10.txt:125
jarg422.txt:119
coral10.txt:105
utrkj10.txt:100
stjlp10.txt:96
hwap10.txt:91
mloss10.txt:90
Time taken for execution0.559
```

Execution time of a single word
0.559 ms
0.624 ms
0.553 ms
0.422 ms
0.558 ms

From the above details we can observe that the minimum time taken by the algorithm to search for a single word Is 0.422 ms

Performance of an algorithm when given two words

```
Example: Search for words: blue fish
truck10.txt:3761
rtlvs10.txt:315
8mpcb10.txt:304
cnfbc10.txt:234
wldfl10.txt:222
moby11.txt:213
2000010a.txt:186
scckg10.txt:178
geogy10.txt:177
whewk12.txt:171
Time taken for execution1.239
```

Execution time of two words
1.239 ms
1.432 ms
1.112 ms
1.876 ms
1.674 ms

From the above table we can observe that minimum time requires dor the execution of two words is 1.112 ms

Execution time for three words

```
Example:
Search for words: blue fish dress
truck10.txt:3761
rtlvs10.txt:315
8mpcb10.txt:304
cnfbc10.txt:234
wldfl10.txt:222
moby11.txt:213
2000010a.txt:186
scckg10.txt:178
geogy10.txt:177
whewk12.txt:171
Time taken for execution2.239
```

Execution time for three words

2.239 ms
2.221 ms
2.114 ms
2.556 ms
2.765 ms

From the above details we can observe that the minimum time taken for the execution of three words is 2.114 ms

Four words:

Example:

```
Search for words: blue fish eyes dress
truck10.txt:3839
whewk12.txt:1175
jbunc10.txt:943
j2ahc10.txt:887
shlyc10.txt:537
gdgdm10.txt:513
grimm10a.txt:481
marie10.txt:479
bulah10.txt:456
prncg10.txt:401
Time taken for execution2.659
```

Execution time for four words
2.659 ms
2.554 ms
2.778 ms
2.983 ms
2.532 ms

Minimum time required for the execution of four words is 2.532 ms

Execution time for five words:

Example:

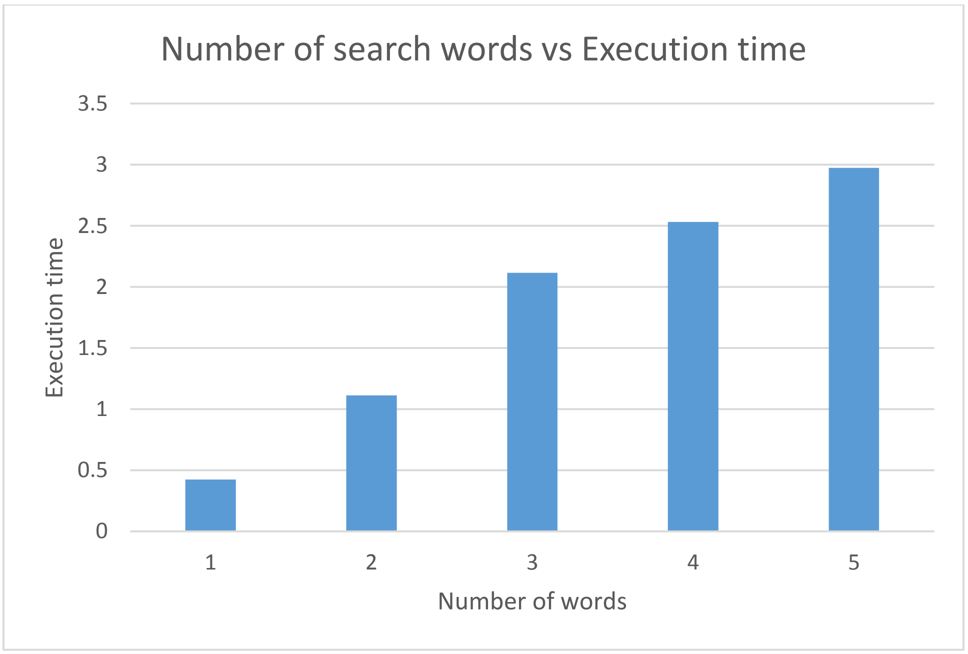
```
Search for words: blue fish eyes dress eat
truck10.txt:3843
jbunc10.txt:1251
whewk12.txt:1249
j2ahc10.txt:911
grimm10a.txt:696
shlyc10.txt:564
```

```
gdgdm10.txt:525
marie10.txt:488
bulah10.txt:472
bough11.txt:465
Time taken for execution2.975
```

Execution time for five words
2.975
3.145
3.098
2.995
3.321

The minimum time required for the execution of five words is 2.975 ms

Number of words searched	Minimum time taken for execution
1	0.422 ms
2	1.112 ms
3	2.114 ms
4	2.532 ms
5	2.975 ms



From the above plot we can observe that as the number of words increases the time required for retrieval increases. This is because as the number of words increases the number of files to be read increases as words starting with different alphabets are in different files. Hence the execution time is directly proportional to the number of words given for search.

References:

http://pstl.cs.uh.edu/resources/shark
http://www.bigdataspeak.com/2013/02/hadoop-how-to-do-secondary-sort-on_25.html
http://www.quora.com/What-is-secondary-sort-in-Hadoop-and-how-does-it-work