Big Data Coursework (M) Report

Methodology

The task is to index Wikipedia files (as of mid-June 2014), English version. MapReduce is used to perform this task. We are given parsed versions of the Wikipedia files, each of which starts with the title is given within double square brackets. Hadoop's default strategy is used for splitting the files – an HDFS block per mapper. The output is a postings list file and a file containing document lengths. We also have to calculate the total number of documents,

Our mapper performs the bulk of the job. The reducer is used only to sort the keys and then print the two files. Let us discuss in details the roles performed by both of them.

Mapper:

Input: One document at a time. The record reader reads the document separated by "\n[[". However the type is LongWritable, Text.

Output: The type is CompositeKeyWritable, NullWritable

a) Key (Composite) -> {term, document id, frequency}; Value -> null

b) Key(Composite) -> {document id, document length}; Value -> null

- Firstly, in each document, the punctuations are removed. The words are then tokenized. Next, stopwords are removed. Remaining tokens are stemmed. Length of the document is incremented every time a non-stopword token is encountered.
- For each non-stopword stemmed token, their occurrence is added to a hashtable. This hashtable contains terms and their corresponding frequency in the current document.
- After reaching the end of a document, the mapper first outputs each term, document id, and its frequency as a composite key. Next, it outputs the document id and document length as a composite key. In the second type, a value "|@|" is added at the beginning of the key as an identifier, so that the reducer can recognize which type of key-value pair is it.
- It also emits two counters : NUM_DOCS and DOC_LENGTH. NUM_DOCS is incremented by 1 per mapper process which is essentially one document read by the mapper at a time, while DOC_LENGTH is incremented by 1 every time the term read from the file is not a stopword.

A grouping comparator is used to group using the natural keys (term for 1sttype of composite key and document id for 2ndtype of composite key), instead of the composite keys.

The partitioner is used to send all keys with the same term to the same reducer.

Reducer:

Input: Type is CompositeKeyWritable, NullWritable.

a) Key (Composite) -> {term, document id, frequency}; Value -> null

b) Key (Composite) -> {document id, document length}; Value -> null

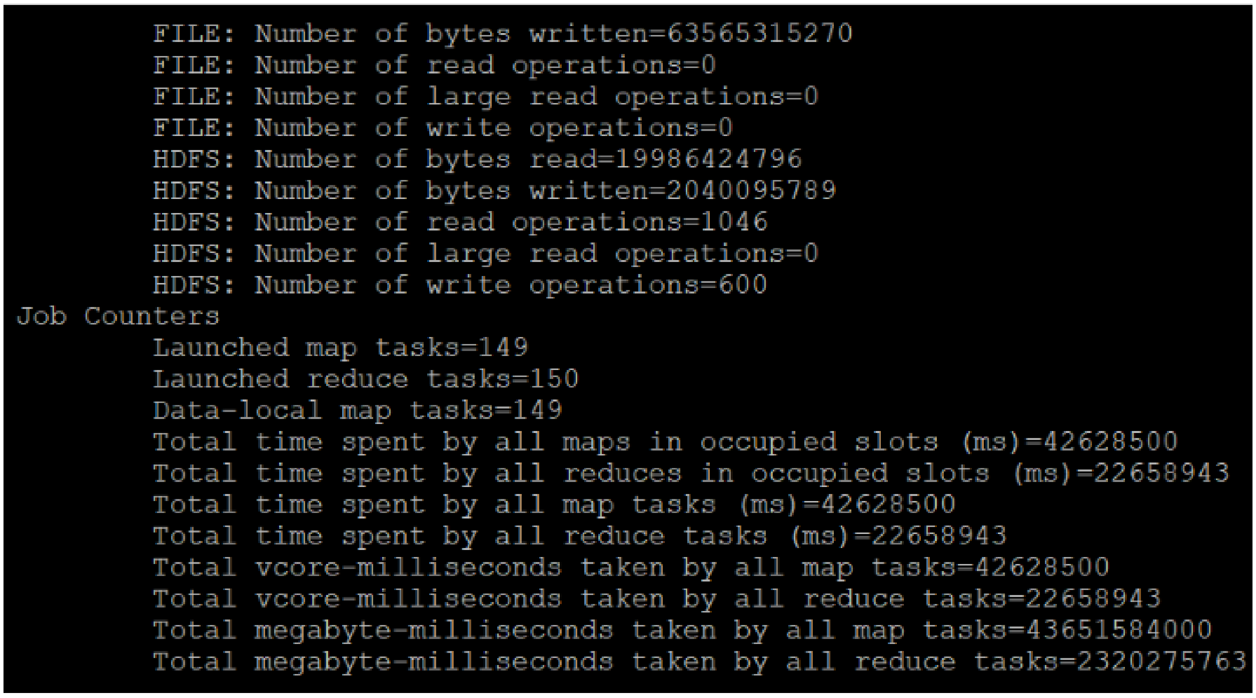
Output: Type is Text, NullWritable.

a) A postings list file

b) A file containing length of each document

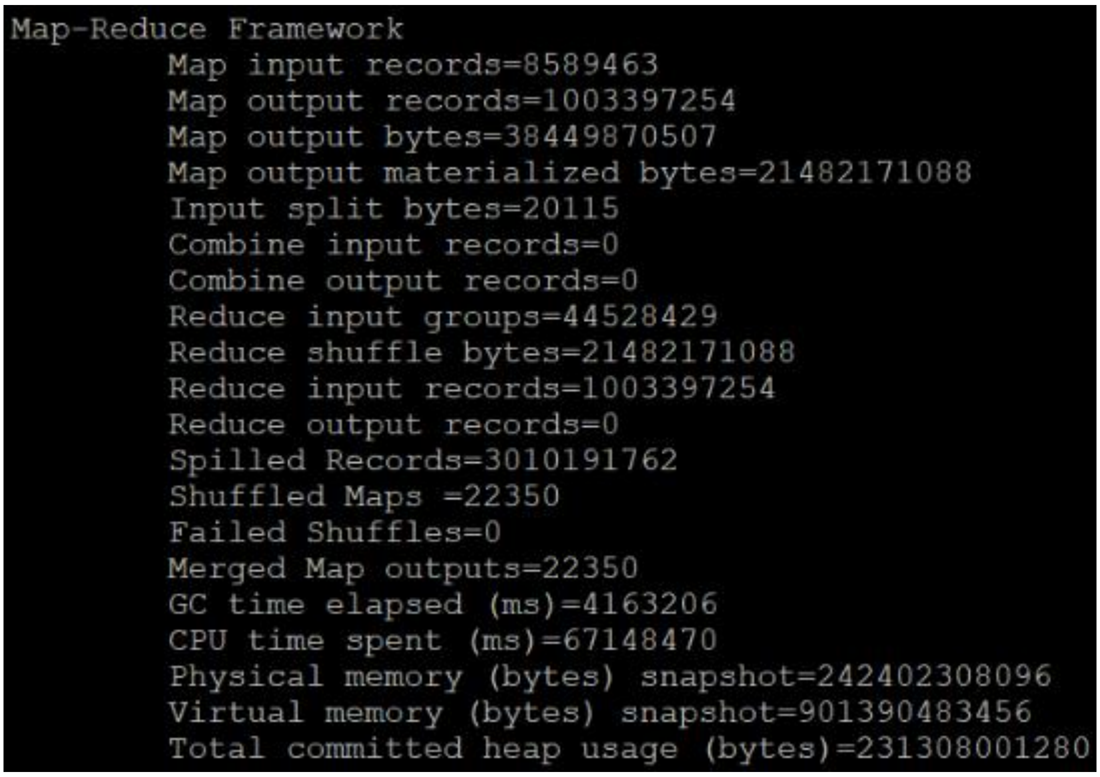
- The reducer receives the composite keys and checks their type.
- If it is for document length, the reducer updates the doc-length file accordingly.
- If term list is obtained, it is sorted in descending order based on the term frequency. The sorted list is then updated to a postings list file.

Results and Discussion



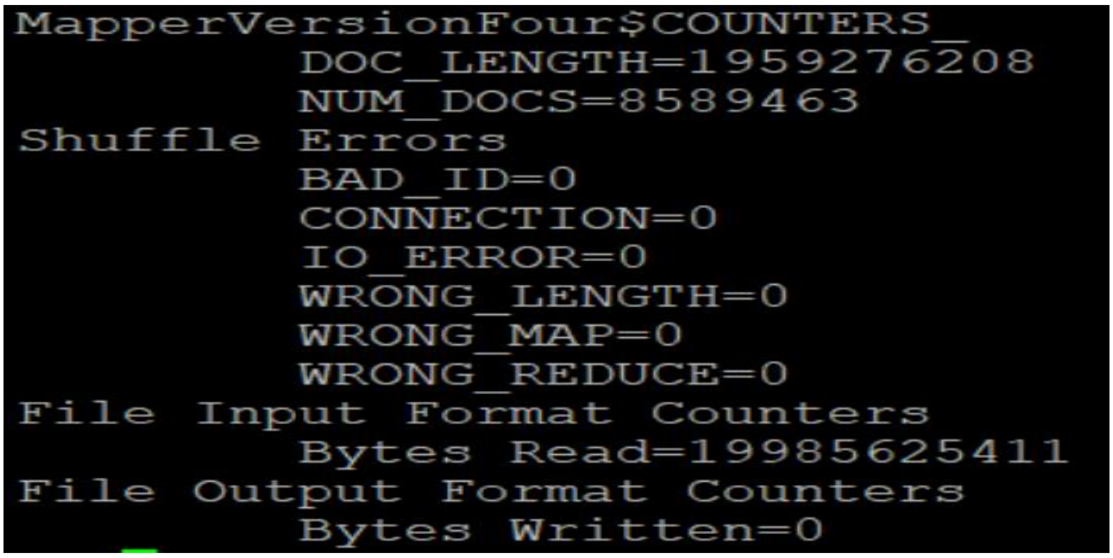
```
        FILE: Number of bytes written=63565315270
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
        HDFS: Number of bytes read=19986424796
        HDFS: Number of bytes written=2040095789
        HDFS: Number of read operations=1046
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=600
Job Counters
        Launched map tasks=149
        Launched reduce tasks=150
        Data-local map tasks=149
        Total time spent by all maps in occupied slots (ms)=42628500
        Total time spent by all reduces in occupied slots (ms)=22658943
        Total time spent by all map tasks (ms)=42628500
        Total time spent by all reduce tasks (ms)=22658943
        Total vcore-milliseconds taken by all map tasks=42628500
        Total vcore-milliseconds taken by all reduce tasks=22658943
        Total megabyte-milliseconds taken by all map tasks=43651584000
        Total megabyte-milliseconds taken by all reduce tasks=2320275763
```

Fig 1. Job Counters



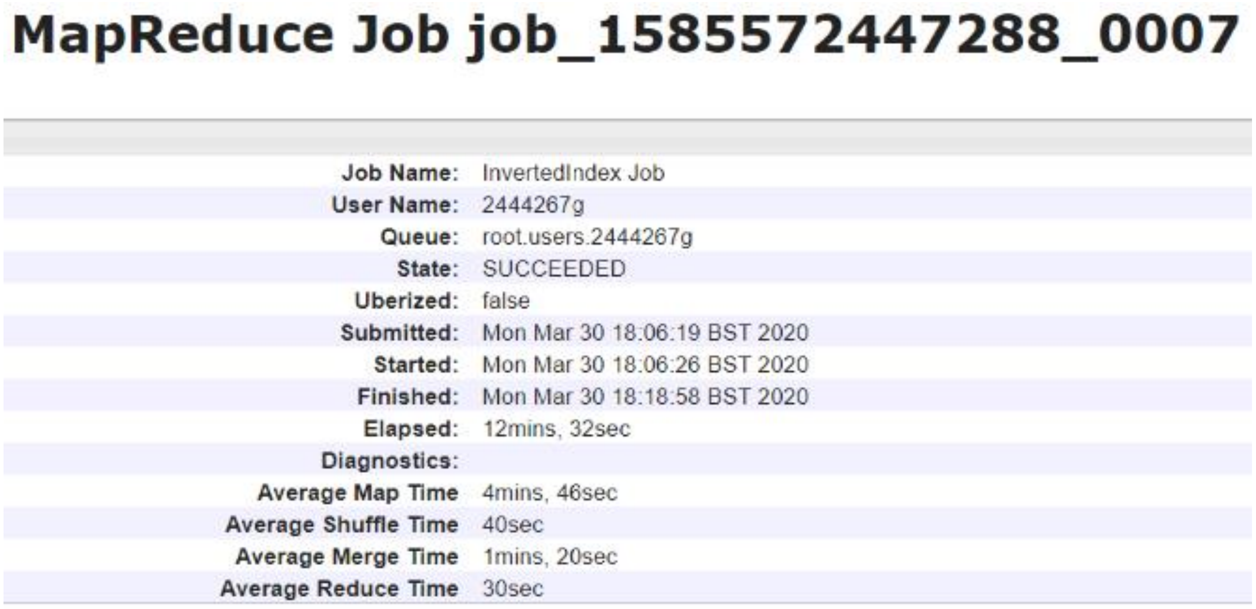
```
Map-Reduce Framework
        Map input records=8589463
        Map output records=1003397254
        Map output bytes=38449870507
        Map output materialized bytes=21482171088
        Input split bytes=20115
        Combine input records=0
        Combine output records=0
        Reduce input groups=44528429
        Reduce shuffle bytes=21482171088
        Reduce input records=1003397254
        Reduce output records=0
        Spilled Records=3010191762
        Shuffled Maps =22350
        Failed Shuffles=0
        Merged Map outputs=22350
        GC time elapsed (ms)=4163206
        CPU time spent (ms)=67148470
        Physical memory (bytes) snapshot=242402308096
        Virtual memory (bytes) snapshot=901390483456
        Total committed heap usage (bytes)=231308001280
```

Fig 2. Map-Reduce Framework



```
MapperVersionFour$COUNTERS_
        DOC_LENGTH=1959276208
        NUM_DOCS=8589463
Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
        WRONG_REDUCE=0
File Input Format Counters
        Bytes Read=19985625411
File Output Format Counters
        Bytes Written=0
```

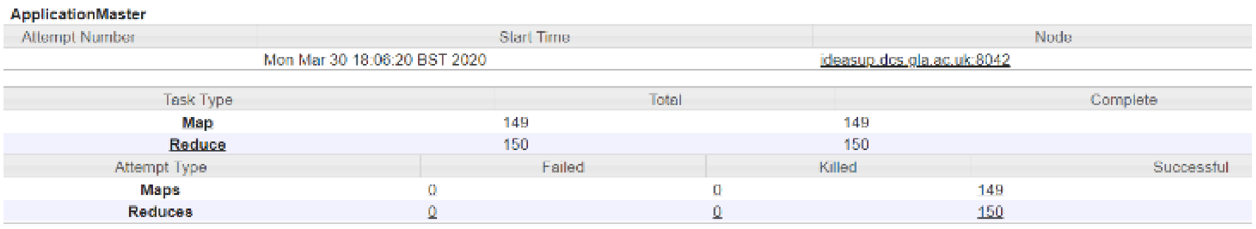
Fig 3. Mapper Counters and Shuffle Errors



MapReduce Job job_1585572447288_0007

Job Name:	InvertedIndex Job
User Name:	2444267g
Queue:	root.users.2444267g
State:	SUCCEEDED
Uberized:	false
Submitted:	Mon Mar 30 18:06:19 BST 2020
Started:	Mon Mar 30 18:06:26 BST 2020
Finished:	Mon Mar 30 18:18:58 BST 2020
Elapsed:	12mins, 32sec
Diagnostics:	
Average Map Time	4mins, 46sec
Average Shuffle Time	40sec
Average Merge Time	1mins, 20sec
Average Reduce Time	30sec

Fig 4. MapReduce Job summary



ApplicationMaster

Attempt Number	Start Time	Node
	Mon Mar 30 18:06:20 BST 2020	ideasup.dcs.gla.ac.uk:8042

Task Type	Total	Complete
Map	149	149
Reduce	150	150

Attempt Type	Failed	Killed	Successful
Maps	0	0	149
Reduces	0	0	150

Fig 5. MapReduce Job summary (continued)

Discussion:

The complete MapReduce task was completed in 12 mins 32 secs. This is quite satisfactory as a huge number of documents (8589463 docs) had to be indexed. The job of the reducer was just to output (and update) two files. As a result, its average execution time is as low as 30secs. On the contrary, the partitoner had to implement a hash table to decide which key-value pair is to be sent to which reducer. Thus, the average merge time was relatively higher – 1mins, 20 secs. 149 mapper tasks and 150 reducer tasks were initiated and none of them failed. All the read and write operations were performed in the HDFS.

Also, **Average document length** is found to be **228.102.** This was calculated on the driver code by utilizing the global counters NUM_DOCS and DOC_LENGTH.

Scalability/ Performance Issues and Proposed Solutions

- The custom partitioner uses a hash function to decide which key goes to which reducer. It is also dependent on the type of the key. Thus basically, if a term is highly prevalent, it may overload a reducer. This was evident during our experimentation. We started with 60 reducers only, and it failed at a very early stage. We increased the number to 150, but faced a failure at 80%. We almost got the job done with 180 reducers but ultimately failed at 99%. The task got finally completed when we used 210 reducers. The partitioner is clearly restricting the scalability.
- Initially, the raw value from the partitioner was used and it worked fine on sample data. But, when we used it on main data, a lot of negative values were obtained. So, we started taking the absolute value of the partitioner.
- Initially, we were using an arraylist to store the stopwords. This was resulting in excessive computation time O(n), which ultimately led to time out in the cluster. Hence, we started using a hash set to store the stopwords, as suggested by our professor, Dr. Nikos Ntarmos. The time complexity has been reduced to O(1) now.
- We also used a lot of replace functions in our code, which takes up a lot of space in memory. This can be optimized by using the StringUtils function in Java which can be **considered in the future**.

References

[1] Many ideas taken from the helper codes provided by Dr Nikos Ntarmos; UOG-Big Data skeleton codes [Java files]; 2020

[2] Distributed Cache concept taken from https://gist.github.com/need4spd/4584416

[3] Custom Writable concept ideas taken from https://javadeveloperzone.com/hadoop/hadoop-create-custom-value-writable-example/

[4] Grouping Comparator and custom partitioner ideas taken from https://www.javacodegeeks.com/2013/01/mapreduce-algorithms-secondary-sorting.html