NoSQL Assignment 2

Deepkumar Patel IMT2021011
Ricky Ratnani IMT2021030
Shrey Salaria IMT2021087

April 2024

Question 1

Part A

We have to find the top 50 word frequency. We describe the approach as follows. The output format is a list of tuples of the form {word, count}.

1. First we utilize the distributed caching option within Hadoop and create a HashSet of the stopwords. .
2. We change the case of the line as per the input arguments.
3. We split the line in words and replaced the stopwords with an empty string using regular expressions.
4. Key is a string and Value is an Integer.
5. For each word we emit {word, 1}. as it represents that the word has occured.
6. For the reducer we accumulate (summation) the value for each word and get a partial result for {word, frequency}.
7. Since we need the top 50 words, we maintain a priority queue in the reducer class to help us get the top values. The tuples (word, count) are sorted in the order that a higher count gets a higher priority and in case of the same count, the lexicographically smaller string gets more priority.

Part B

We need to find the cooccurrence of the top 50 words obtained in the previous part. We describe the approach as follows. The output format is a list of tuples of the form {word-1, word-2, count}.

1. First we utilize the distributed caching option within Hadoop and create a HashSet of the top 50 words.

2. We change the case of the line as per the input arguments.

3. We split the line in words and use a two-pointer approach to iterate over the pair of words having a predecided distance d between them. we check if both the words belong to the HashSet and then emit as follows.

4. the key is a pair of words and the value is an integer.

5. we emit {{word-1, word-2}, 1}. the one shows that the two words occur together. we also make sure that word-1 is lexicographically smaller than word-2, to prevent permutations. This will eventually help reduce the total memory used for the output file.

6. For the reducer we accumulate (summation) the value for each word pair and get a partial result for {{word-1, word-2}, frequency}. which is then saved to an output file.

Part C

Using the stripes approach, we need to find the cooccurrence of the top 50 words. We describe the approach as follows.

1. First we utilize the distributed caching option within Hadoop and create an ArrayList of the top 50 words. We then sort the list in lexicographic order.

2. We change the case of the line as per the input arguments.

3. We split the line in words and use a two-pointer approach to iterate over the pair of words having a predecided distance d between them. we check if both the words belong to the ArrayList and then emit as follows.

4. We represent the key as a string and the value as an array of length n (where $n = 50$). let onehot(i) represent the array with only index i set to 1 and all others set to 0. This will be useful while emitting.

5. for each pair word-1 and word-2, we emit {word-1, onehot(idx-2)} and {word-2, onehot(idx-1)}. where idx-1 and idx-2 are the index at which word-1 and word-2 occur in the arraylist. On emitting {word-1, onehot(idx-2)} is signifies that word-1 has co-occurred with word-2, and since the value at index is set to one, it signifies the current concurrence.

6. For the reducer we accumulate the value for each word and get a partial result of type {word, accumulator}. the accumulator is an array obtained using the element-wise summation of all the values the reducer receives for a key.

7. We then write the word and the accumulator to the output file.

to describe the output file, we have a n x n matrix (say M), such that M_{ij} represents the number of times the i^{th} and j^{th} word have occurred together. The k^{th} word is the word at index k in the array list of the top 50 words when sorted lexicographically.A simple observation is the M is a symmetric matrix. For simplicity while viewing we also have a column of the words in the lexicographically sorted order. an example is shown in table 1.

abc	1	0	1
def	0	2	0
ghi	1	0	3

Table 1: Example 1C output format

Part D

When we optimize map-reduce operations, we try to reduce the computation at the reducer level. We can achieve this by applying the reduce function in the Mapper class to aggregate partial results before emitting key-value pairs. We can aggregate the results using a HashMap. Class Level Optimization is when we define the HashMap as a class attribute, and Function Level Optimization is when we define the HashMap in the scope of the map function in the Mapper Class. This can be seen easily in our code files. Intuitively, class level optimization and function level optimization perform comparably, which are in turn faster than no optimization (obviously). The results for the experiments can be seen in tables 2 , 3, 4, and 5.

dist/opt	no optimization	class level optimization	function level optimization
1	3746	2858	3002
2	3677	2903	2866
3	3833	3185	3078
4	3829	2942	3050

Table 2: Total time spent by all map tasks in Part B

dist/opt	no optimization	class level optimization	function level optimization
1	2126	1587	1527
2	2119	1614	1544
3	2095	1558	1687
4	2047	1502	1537

Table 3: Total time spent by all reduce tasks in Part B

dist/opt	no optimization	class level optimization	function level optimization
1	4648	3400	3429
2	4505	3374	3510
3	4410	3589	3640
4	4752	3634	3510

Table 4: Total time spent by all map tasks in Part C

dist/opt	no optimization	class level optimization	function level optimization
1	2359	1580	1387
2	2219	1397	1432
3	2206	1429	1452
4	2530	1577	1553

Table 5: Total time spent by all reduce tasks in Part C

Question 2

Part A

We have to identify the top 100 words with the maximum document frequency. Document frequency is the number of distinct documents in which the term appears. We describe the approach as follows.

1. First we utilize the distributed caching option within Hadoop and create a HashSet of the stopwords. .
2. We change the case of the line as per the input arguments.
3. We split the line in words and replaced the stopwords with an empty string using regular expressions.
4. Key is a string and Value is a pair of string and integer.
5. We are supposed to use the PorterStemmer in the opennlp module, so whenever We refer to a word, it means a stemmed word.
6. for each word in the line, we emit {word, {doc, 1}}. this signifies the occurrence of the word in the doc.
7. for the reducer, we create a HashSet, and insert all the document IDs in it. HashSet doesn't allow duplicates, meaning the size of the HashSet is the document frequency.
8. Now since we need the top 100 words. just like in question 1 part a. We maintain a priority queue in the reducer class to help us get the top values. The tuples (word, count) are maintained in the order that a higher count gets a higher priority and in case of the same count, the lexicographically smaller string gets more priority.
9. We then output the word and document frequency obtained in the previous step in a file.

Part B

1. First we utilize the distributed caching option within Hadoop and create an ArrayList of the top 100 words from part A. We then sort the list in lexicographic order.

2. We also create a HashMap (say dfmap) which stores the document frequency simultaneously with the first step.

3. We change the case of the line as per the input arguments.

4. We are supposed to use the PorterStemmer in the opennlp module, so whenever We refer to a word, it means a stemmed word.

5. We represent the key as a string and the value as an array of length n (where $n = 100$). let onehot(i) represent the array with only index i set to 1 and all others set to 0. This will be useful while emitting.

6. for each word we emit {docId, onehot(i)}, where i is the index at which the word is found in the sorted arraylist. This represents that the word at index i is found in the document.

7. For the reducer we accumulate the value for each word and get a partial result of type {word, accumulator}. the accumulator is an array obtained using the element-wise summation of all the values the reducer receives for a key.

8. Note, the value in the accumulator at index i is the term frequency for the word at index i in the sorted ArrayList to the document.

9. We have the document frequency from the map, we also have the term frequency from the accumulator, hence we calculate the score and output it to a file.

10. Since we are using the stripe approach the output is similar to a matrix M of shape [number of docs, 100]. so we write a Python script to convert the output to a TSV file format.

1 Errors And Experimentation Results

The most common error is the Java heap space error. this is why we decided to work with a subset of 2500 files out of 10k files available in the Wikipedia dataset.

Note: We have also provided all the messages (from the terminal) and the map-reduce result files in our submission files. We have included all the experiments like trying out various values of d in question 1.

```
2024-04-07 14:27:27,353 INFO client.DefaultNoHARMFailoverProxyProvider: Connecting to ResourceManager at /127.0.0.1:8032
2024-04-07 14:27:27,578 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your application with ToolRunner to remedy this.
2024-04-07 14:27:27,592 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/hdoop/.staging/job_1712476972137_0003
2024-04-07 14:27:27,801 INFO input.FileInputFormat: Total input files to process : 1
2024-04-07 14:27:27,840 INFO mapreduce.JobSubmitter: number of splits:1
2024-04-07 14:27:27,990 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1712476972137_0003
2024-04-07 14:27:27,990 INFO mapreduce.JobSubmitter: Executing with tokens: []
2024-04-07 14:27:28,113 INFO conf.Configuration: resource-types.xml not found
2024-04-07 14:27:28,114 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2024-04-07 14:27:28,161 INFO impl.YarnClientImpl: Submitted application application_1712476972137_0003
2024-04-07 14:27:28,183 INFO mapreduce.Job: The url to track the job: http://ricky-HP-Pavilion-Laptop-14-dv0xxx:8088/proxy/application_1712476972137_0003/
2024-04-07 14:27:28,184 INFO mapreduce.Job: Running job: job_1712476972137_0003
2024-04-07 14:27:33,257 INFO mapreduce.Job: Job job_1712476972137_0003 running in uber mode : false
2024-04-07 14:27:33,258 INFO mapreduce.Job:  map 0% reduce 0%
2024-04-07 14:27:47,356 INFO mapreduce.Job: Task Id : attempt_1712476972137_0003_m_000000_0, Status : FAILED
Error: Java heap space
2024-04-07 14:28:00,423 INFO mapreduce.Job: Task Id : attempt_1712476972137_0003_m_000000_1, Status : FAILED
Error: Java heap space
2024-04-07 14:28:14,488 INFO mapreduce.Job: Task Id : attempt_1712476972137_0003_m_000000_2, Status : FAILED
Error: Java heap space
2024-04-07 14:28:31,563 INFO mapreduce.Job:  map 100% reduce 100%
2024-04-07 14:28:31,570 INFO mapreduce.Job: Job job_1712476972137_0003 failed with state FAILED due to: Task failed task_1712476972137_0003_m_000000
Job failed as tasks failed. failedMaps:1 failedReduces:0 killedMaps:0 killedReduces: 0

2024-04-07 14:28:31,627 INFO mapreduce.Job: Counters: 13
        Job Counters
                Failed map tasks=4
                Killed reduce tasks=1
                Launched map tasks=4
                Other local map tasks=3
                Data-local map tasks=1
                Total time spent by all maps in occupied slots (ms)=49514
                Total time spent by all reduces in occupied slots (ms)=0
                Total time spent by all map tasks (ms)=49514
                Total vcore-milliseconds taken by all map tasks=49514
                Total megabyte-milliseconds taken by all map tasks=50702336
        Map-Reduce Framework
                CPU time spent (ms)=0
                Physical memory (bytes) snapshot=0
                Virtual memory (bytes) snapshot=0
```

Figure 1: Java Heap Space Error