CS 839 NoSQL Systems: Assignment 2

Chinthan Chandra
IMT2020109
IIIT Bangalore
chinthan.chandra@iiitb.ac.in

Kaushik Mishra
IMT2020137
IIIT Bangalore
kaushik.mishra@iiitb.ac.in

Vivek Tangudu
IMT2020110
IIIT Bangalore
vivek.tangudu@iiitb.ac.in

INTRODUCTION

We were given two problems to be solved using `MapReduce` and `Apache Hadoop`. The first problem is to find the count of each part-of-speech tag. The second problem is to find document frequency and then inverse document frequency.

MAPREDUCE APACHE HADOOP

9 points

Problem 1. PART-OF-SPEECH TAGGING USING PAIRS AND STRIPES.

Each word in a sentence falls into one of several distinct grammatical categories, referred to as parts of speech. The process of assigning words in a sentence to their appropriate part of speech, such as a verb, noun, adjective, etc., is known as part-of-speech tagging, or POS tagging. Employ the MapReduce idea to fulfill the following requirements.

- **From all words in the `Wikipedia-EN-20120601ARTICLES.tar.gz` corpus, calculate the count of each POS tag (AUX, PRON, VERB, PRON, ADP, PROPN, PUNCT, NOUN, ...) across all the articles using a MapReduce job.**
- **Create two MapReduce programs: one that uses the Pairs approach and the other that uses the Stripes strategy. Compare the execution time of your programs.**

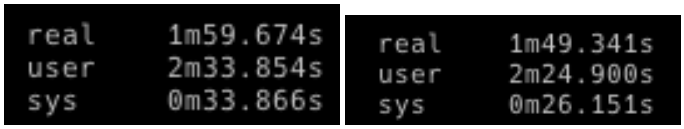
Note: To identify a possible set of POS tags, you may use the OpenNLP library and a pre-trained POS tag model, which are available from `opennlp-tools-1.9.3.jar` and `opennlp-en-ud-ewt-pos-1.0-1.9.3.bin` on LMS. You may check the PosTag.java file on the LMS to learn more on how to use the OpenNLP library. Instructions for running PosTag.java and the required additional jar files are also provided. While running your hadoop job from the terminal, include the `opennlp-tools-1.9.3.jar` to the class path as shown here, `HADOOP_CLASSPATH` environment variable, as follows: `export HADOOP_CLASSPATH="<PATH-TO-LIBRARIES>/*"`

Ans. For this question, we have to compare performance between pairs and stripes strategies. The initialization for both these is the same. We get the pre-trained POS tag model that is given to us that automatically assigns the correct grammatical categories to each word. We load the model and create a simple tokenizer. Then we take the article as input and then tokenize it into words. After this POS tagging is done for each token. The steps up until here are the same for both approaches.

1 **Pairs Approach** In this approach we emit every key-value pair formed in the mapper. After being tokenized and tagged, we iterate through the tags array and emit $< tag, 1 >$ for each element in the tags array. The reducer then just sums the counts which belong to the same key. So we have got the count of each POS tag in the articles.

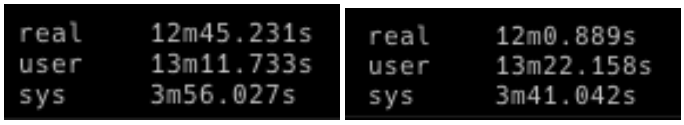
2 **Stripes Strategy** Instead of emitting every key-value pair found, we aggregate these in a hashmap i.e an associative array for every article. Since we require the POS tag counts of all documents, we emit the key to be the same for every article. This is what is done in a stripes approach, we emit a key-value pair, where the value is an associative array, in our case it is the key-value pair of each POS tag in that article. In the reducer, all articles' count of POS tag is stored as a list of iterable Map objects. Here, we create another associative array, a final one, which stores the count of each POS tag over all articles by first iterating through the list of map objects and then adding the key and value if it does not exist, if it does just increase the count. For emitting the values, it iterates through the final associative array and emits the key-value pair, which is $< POS\_tag, total\_count >$.

We ran both programs for 50 articles initially and the stripes approach was faster than the pairs. We also tested with 200 and 1000 articles. We didn't try higher than this because of system limitations. The time comparisons of both are shown in Figure 1 and 2. We can clearly conclude that stripes approach is faster, which in accordance to what was learnt in class.



(a) Pairs (b) Stripes

Fig. 1: POS tagging using MapReduce Comparison for 200 articles.



(a) Pairs (b) Stripes

Fig. 2: POS tagging using MapReduce Comparison for 1000 articles 1).

RUNNING KEYWORD QUERIES OVER WIKIPEDIA

Problem 2. INDEXING DOCUMENTS VIA HADOOP

1 **Implement a pair of a Map and a Reduce function which, for each distinct term that occurs in any of the text documents in `Wikipedia-EN-20120601_ARTICLES.tar.gz`, counts the number of distinct documents in which the term appears. We will call this value the Document Frequency (DF) of that term in the entire set of Wikipedia articles. Store the resulting DF values of all terms in a single TSV file with the following schema:**
`TERM<tab>DF`
While generating the output in the above format, consider filtering out all terms that belong to the stopwords.txt file shared on LMS. (You may perform this filter operation in your map method.) Identify the top 100 terms with a high document frequency. Use those terms alone for the next subproblem.
Ans.
The goal is to find the document frequency of each term, i.e in how many documents a term occurs. By the definition, it is clear that if we have multiple occurrences of the same word/term we have to count it as 1. Our approach followed the same initial steps until tokenizing as we did in Problem 1 after that, we created a set and added all the tokenized words to this set. Now we iterate through this set, check if they belong to the set of stopwords files we have given in the command file argument, and emit key-value pairs if they are not stopwords. We had two approaches, either emit $< word, 1 >$ or $< word, filename\_of\_article >$.
Either of them mean the same, that a term consists in one article. In the reducer, we just need to sum the number of 1's in the values field or take the count of the number of filenames. Since we also want to keep only 100 terms, we have kept the highest score, we can change this to the lowest 100 scores or choose 100 random terms too.
The implementation we did for this using a TreeMap, which uses the score as a key and term as a value and stores the key-value pair sorted based on the key. If the length exceeds 100, we remove the first entry which is the lowest-scored term. And we emit in the cleanup phase of the reducer, i.e once all the reductions are done, we emit the top 100 scored terms from the reducer class. This is filtering on the output.
Now since we also need to store the output in a particular format, i.e tab-separated values, we configure the same. This configuration is done by setting the `mapreduce.output.textoutputformat.separator` configuration as "\t" and setting the `OutputFormatClass` as `TextOutputFormat.class`. We can use this to configure the separator between the key and value in the output text file. So the output key-value pair is $< term, score >$.

2 **Implement another pair of a Map and a Reduce function which, for each document in `WikipediaEN-20120601_ARTICLES.tar.gz`, first counts the number of occurrences of each distinct term within the given document. We will call this value the Term Frequency (TF) of that term in the given document. Implement a stripes algorithm here as we are dealing with just 100 terms.**
In a second step, your combined MapReduce function should multiply the TF value of each such term with the inverse of the logarithm of the normalized DF value calculated by the previous MapReduce function, i.e., `SCORE = TF×log(10000/DF+1)` for each combination of a term and a document. You may cache the former TSV file with the DF values by adding it via `Job.addCacheFile(<path-to-DF-file>)` in the driver function of your MapReduce program. The Map class should then load this file upon initialization into an appropriate main-memory data structure.
The result of this MapReduce function should be a single TSV file that has the same schema as the file we used in the previous exercise sheets:
`ID<tab>TERM<tab>SCORE`.

Ans. From the document frequencies calculated in the previous problem, we need to change the file name and add `.tsv` at the end. We can send this as input to our MapReduce job from the command line by writing the location of the file after `-idf`. The approach here is to load the document frequency map from the cache files in both mapper and reducer. In the mapper after doing the initial steps until stemming as done in problem 1, we then go ahead and check if the token is in the loaded document frequency map, if it exists we add it to the associative array of the mapper with the term as key and value as the 1, if the term does not exist in the associative array if it does, we increase the value by 1. The mapper emits

`<article\_id,Map<term,frequency>>`

as the output article_id acts as the identifier for each article, we have used filename as the article_id. Since we are using the stripes approach the output is an associative array or map.
In the reducer, we have the emitted key-value pair, if we assume that there is no duplication of files then we can directly iterate over one associative array and emit outputs. But we have iterated over each associative array that has the same article_id. We accumulate the term frequencies by creating a final associative array. After this, we iterate through the associative array and then get the document frequency pertaining to the term which is the key, and then calculate the Inverse Document Frequency for each term. We emit this score as the value and the key as a string of fileName i.e the article_id and the term joined together with a tab space between them. This gives the desired output. Due to memory limitations, we could not run on all files, we have tried for 50, 200, 1000, and 5000 articles.