UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG
SCHOOL OF ELECTRICAL AND INFORMATION ENGINEERING

ELEN4020A: Data Intensive Computing: Lab 3

Sbonelo Mdluli(1101772), Heemal Ryan(792656), Haroon Rehman(1438756)

April 16, 2019

1 INTRODUCTION

MapReduce is a programming paradigm for facilitating salable computations across Hadoop clusters while also aiding in the decomposition of problems so that they may be solved more easily and faster. It is commonly used in big data problems where there is much data to be processed and also is able to be divided among many processors, whether on a single machine or on a cluster. It is not appropriate for tasks that cannot be divided easily, as it can be quite cumbersome to split and combine the outputs, or the process inherently cannot be divided. The following report showcases and discusses an implementation of MapReduce (Mrs-MapReduce) as per the ELEN4020A Lab 3 brief and consists of the following sections: Overview of the MapReduce process, code structure which is written in Python as per the requirement of the MapReduce framework being used and finally results and analysis on the output and performance of the presented solution. The appendix contains pseudo-code for the implemented algorithms.

2 MAPREDUCE PROCESS

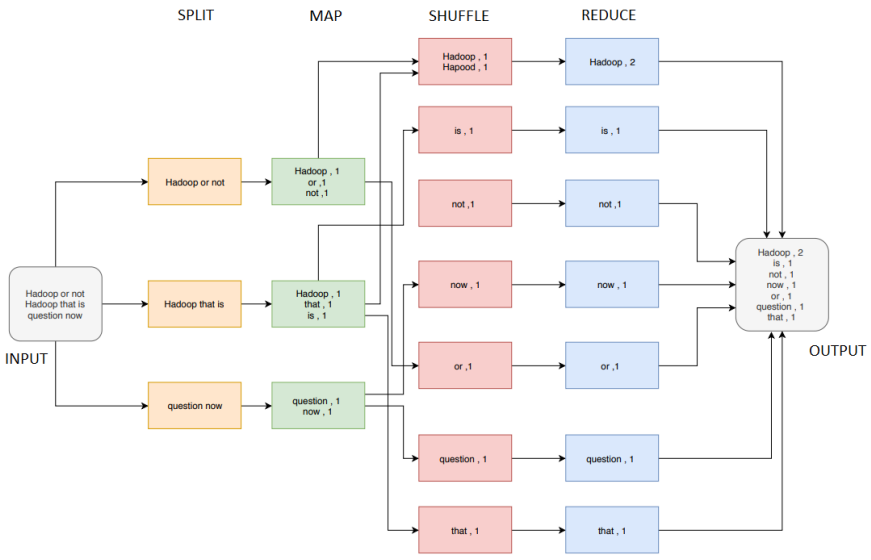

Figure 1: MapReduce process

2.1 Summary at each stage

- Split
  The process starts by distributing work among all available map nodes.
- Map
  Next, words are mapped and given a hard-coded value of "1" (this is done so as to count the number of times a word occurs)
- Shuffle
  During the shuffle phase, like words (keys) are separated and grouped together.
- Reducer
  The reducer sums the values within these individual groups of keys to produce a final output, that being, the number of times unique words appeared in text.

3 CODE STRUCTURE

The following subsections give details about the algorithms implemented using the MapReduce framework. The framework chosen is Mrs.Mapreduce and thus requires to be coded in python. The code is centralized around the map and reduce operations that are provided by the framework, as is the focus the lab.

3.1 Word count

The word count algorithm returns the frequency of each unique word within a piece of text while ignoring certain Stop words (this refers to commonly used words within a language e.g. for, as, the, etc).
Words were considered case-insensitive, this was done by converting all words to lower case before counting the number of times they appear. The code essentially maps each word to a value (which is always 1) and then shuffles and groups similar words in each process. Finally, the value associated with words with the same key are summed in order to obtain the total number of times that each word is seen in the text. The list of key value pairs are written to a text file for further processing.

3.2 Top-K query

This piece of code returns the K most frequent words in a piece of text (K=10,20) while ignoring stop words. The top query requires the output of the word count algorithm. The text file produced by the previous algorithm is read and then sorted in descending order by value. The output is then the top K words and their frequencies that have been queried.

3.3 Inverted index

The final algorithm concerns an inverted index. In computer science, inverted index can be understood as a database index that stores a map to content, such as words or numbers. The benefit of using inverted index is that it allows for fast full-text searches.
The implementation consists of listing each word with the line numbers that the word occurs. The inverted text algorithm makes use of a list to store page numbers, the constant value "50" is used to skip the first 50 lines of a file as it may include headings, dates, etc.

This algorithm consists of two processes. Firstly, instead of mapping each word to a value of

1, it is mapped to the line in which it occurs. This is done using the map and reduce functions in Mrs.MapReduce. In the reduce phase, the line numbers are not summed, but merely listed next to each word. Similarly the stop words as in the word count algorithm are ignored as well as digits occurring in the text. The output is then written to a text file for further processing.
The next part involves reading the output from the map-reduce operations and then just formatting and selecting which part of the inverted index should be shown.

4 Results and Analysis

It was found through testing that all the algorithms worked as intended. The word count algorithm was able to successfully determined the frequency of unique words taking into consideration case sensitivity and ignoring stop words.
The top query algorithm successfully produced the top 10 and top 20 most frequent words.
The inverted index algorithm was able to correctly map words in text to the line numbers they appeared on. The algorithm produces more than 50 lines where some words occur, which can be fixed.

Contained within the Appendix are snippets of the program output.

5 Conclusion

MapReduce iss designed to solve a major modern day problem, processing very large data sets, MapReduce is able to help solve this issue while also hiding the details of parrallelizing workflow and code. It has the responsibility of executing written map and reduce functions in a distributed environment .
Mrs-MapReduce is one of many available MapReduce frameworks however from experience, Mrs-MapReduce proved to be an easy to use framework as well as being simple to install and configure. The report discusses three algorithms that use the MapReduce framework, namely; word count, top-k query and a modified inverted index. The algorithms perform as required on large text, thus showing how MapReduce is useful for sorting huge sets of data.

References

[1] Pythonhosted.org. (2019). User Guide — Mrs MapReduce v0.9 documentation. [online] Available at: https://pythonhosted.org/mrs-mapreduce/user_guide.html [Accessed 16 Apr. 2019].

A Appendix

A.1 Pseudo-code Lab 3

Algorithm 1: Simple word count excluding stop words

```
definition map(self,key,value):
stop = [insert words to be excluded]
value = value.split()
value = [word.strip(string.punctuation).lower()]

for length of stop do
    while stpWord in value do
        value.remove(stpWord)
    end
end
for word in value do
    if not str.isdigit(word) then
        yield(word,1)
    end
end

definition reduce(self,key,values):
vals = sum(values)
yield vals
```

Algorithm 2: Top K-query

Makes use of Algorithm 1

```
definition sortList(text):
text.sort(key= lambda x: x[1], reverse=True)
return text

definition top_Kquery(text, k):
return(text[:k])

definition process():
cur_ = os.getcwd()
dir_ = os.path.join(cur, "outdir")
fname = os.path.join(dir_,"source_0_split_0_.mtxt")
f = open(fname, "r")
text = f.readlines()
for i in text do
    text[text.index(i)] = tuple(i.split())
end
return text

definition main()
sorted_txt = sortList(process())
query = [10,20]
for k in query do
    print("Top: "+ str(k) + " occuring words")
    for i in top do
        _Kquery(sorted_txt,k)
        print(i)
    end
end
```

Algorithm 3: Algorithm line count

```
definition map(self,key,value):
stop = [insert words to be excluded]
value = value.split()
value = [word.strip(string.punctuation).lower()]
for length of stop do
    while stpWord in value do
        value.remove(stpWord)
    end
end
for word in value do
    line = value[0]
    if not str.isdigit(word) then
        yield (word,line)
    end
end
definition reduce(self,key,values):
vals = sum(values)
yield vals
```

Algorithm 4: Inverted Index

Makes use of Algorithm 3

```
definition lis_distinct(text, k) :
return set(text[:k])

definition process(k):
cur_ = os.getcwd()
dir_ = os.path.join(cur,"outdir")
fname = os.path.join(dir_,"source_0_split_0_.mtxt")
f = open(fname, "r")
text = f.readlines()
for line,i in enumerate(text) do
    if line > 50 and not re.search(r'/ d',i[0] then
        text[text.index(i)] = tuple(i.split())
        print(i)
    end
    if line==50 then
        break
    end
end

definition main()
k = 50
print("First distinct: "+ str(k) + " words")
fil = process(k)
```

A.2 Program Output

```
childish 2
children 127
children's 2
children--and 1
children--are 1
children--for 1
chimera 1
china 1
chines 2
chirping 1
chisel 1
choice 17
choices 1
choose 27
chooser--god 1
chooses 3
choosing 6
chords 1
chorus 5
choruses 2
chose 9
chosen 17
christ 15
christ's 2
christendom 1
christian 23
christianity 3
christians 4
chronique 1
chronological 2
chronology 1
chryses 5
church 18
churches 2
churlish 1
```

Figure 1: Word Count

```
breath ['6016', '10661', '11990', '21559']
breathe ['3355']
breathing ['1522', '11885']
bred ['6711', '15220']
breed ['468', '2698', '6685', '6763', '6893', '6917', '13724',
breeder ['2687', '6708']
breeding ['4419', '4508', '5444', '6679', '15662', '15680', '21
breeds ['21344']
breeze ['6042', '7413', '12587']
brethren ['2811']
bribe ['22814']
bribed ['12975']
bribery ['6261']
bribes ['1531', '1707', '21856']
bricks ['8852']
bride ['7031', '20213']
bridegroom ['3085', '7031', '15826', '17690', '20214']
bridegrooms ['2696', '15727']
brides ['2696', '15727']
bridle ['3090', '3564', '5462', '7485', '17715']
brief ['4647', '5552']
briefly ['2912', '6390', '7898']
bright ['5637', '17733', '21064', '24086']
brighter ['2887', '4165', '5666', '18764', '24139']
brightest ['18785', '19577', '24159']
brightest--the ['5676']
brightness ['18262']
brilliant ['2337', '23377']
bring ['1093', '2691', '2929', '2936', '3109', '3168', '4507',
'14825', '15112', '15726', '15764', '16408', '17421', '17750',
bringing ['835', '4489', '6229', '15738', '16081', '16480']
brings ['3319', '3973', '5136', '6610', '7913', '8108', '10472'
brink ['768']
brittle ['13135']
```

Figure 2: Inverted Index