ELEN 4020 Lab 3: MapReduce and MrJob

Junaid Dawood (1094837), Xongile Nghatsane (1110680), Marissa van Wyngaardt (719804)

***Abstract*—This report discusses typical use cases for the MapReduce programming model. This discussion is informed by the practical use of a MapReduce framework (MrJob) in Python. Specifically, this involves the implementation of three common MapReduce use cases: 1) word count, 2) K-most frequent analysis, and 3) inverted index. A detailed description of these implementations is given within this report as is the associated pseudocode.**

I. Introduction

The MapReduce programming model, as originally defined by Google, allows for massively distributed parallelisation of big data tasks [1]. MapReduce is simply a programming model, and as such, one may either implement the model or make use of existing frameworks which implement the model. The most commonly used model is Apache Hadoop, although there exist viable alternatives such as Pheonix++ (C++) and MrJob (Python) [2]–[4].

Generally, big data tasks involve datasets which are far too large to be hosted on a single system. Thus, MapReduce makes use of a distributed file system (DFS) in order to share these large datasets as chunks, between nodes in a cluster.

This report discusses the implementation of three common MapReduce solutions using the MrJob framework which implements the MapReduce model.

II. Background

MapReduce in its simplest form consists of two operations: Map and subsequently Reduce. The Map operation is supplied with a chunk of input data, from which it emits key-value pairs. There are typically several *mappers* working in parallel, across many nodes each supplied with independent chunks of the same input dataset. In the vast majority of cases, it is the MapReduce framework which will automatically allocate chunks of input data to individual *mappers*.

After the Map phase completes, the emitted key-value pairs are sorted and shuffled, such that all values related with a single key are grouped together, typically achieved through the use of a (distributed) hash table. The Reduce operation then begins, taking in a key and a list of values as its argument and emitting a single key-value pair. Like the Map operation, there are generally several *reducers* working in parallel, each with their own input data. Again, the supplying of input parameters is handled by the framework implementing the MapReduce model. The results emitted by all of the reducers are subsequently combined to form the final output of the MapReduce step.

Both the Map and Reduce operations are defined by user code, so as to specify the processing that occurs prior to the emission of key-value pairs from the Map operations, and to specify the structure of said pairs. The rules for the combination of several values so as to emit a single pair from Reduce operations are also defined by user code.

Generally, MapReduce tasks require more than a single step; as a result, most frameworks allow for the output of one MapReduce step (the collective output of its reducers) to act as the input for another MapReduce step. The MrJob framework allows the user to explicitly declare steps and their respective operations within classes which derive from the MrJob class.

On the topic of operations, there often exists an intermediate operation between Map and Reduce called Combine. This operation is typically used to achieve some intermediate processing to improve performance.

In addition, the MrJob framework allows for the definition of both pre and post operations for each of the three main operations, e.g. *mapper_init* and *mapper_final*. These are typically required for maintaining an auxiliary structure which exists separately from the main MapReduce operations.

III. Word Count

The word count algorithm analyses input data to generate an output table which lists the words present in the input alongside their respective numbers of occurrences. The implemented word count algorithm consists of two steps. The first deals with the removal of 'stop' words from chunks of input data, and the second tallies the occurrences of words to present a list of words and the number of occurrences for each. The list of stop words (NLTK) is attached at the end of this document.

The first step consists of a simple Map operation which emits lines numbers as keys and words as values from an input data chunk, if said words are not a part of a list of stop words. The reduce operation takes the list of words generated for each line-number and creates a single delimited string from the list of words; emitting the line-number and string as a key-value pair.

The use of line-numbers is not necessary for this approach to work; all words could simply be mapped to a single key and the end result would still be correct. However, the use of a single key would mean that only a single mapper is used within the second Map operation.

The second Map operation simply extracts words from a delimited list and emits a key-value pair wherein the key is the word and the value is 1. Thereafter, the word count portion employs an optional Combine operation which emits words and the sum of received 1s as a key-value pair. Each reducer then takes in a list of values for a given

word and emits the word and the sum of the values as a key-value pair.

IV. Top K Words

The Top K words algorithm returns a list of the top K most frequent words in a sample text. The algorithm incorporates a similar approach to the word count algorithm described above. Specifically, the algorithm alters the second Reduce operation of the word count algorithm and specifies an additional step which contains only a Reduce operation.

The altered Reduce operation emits all *(word,count)* pairs as tuple values, using the same key (NULL) for all. Thus the combined output of all reducers is a single key associated with a list of *(word,count)* tuples. Of course, this devolution to a single key implies the use of a single reducer within the next step.

The Reduce operation in the additional step then selects the top K words from the list of tuples based on the number of occurrences specified in the tuples. This is done using the built-in Python *heapq.nlargest* function.

In addition, the configuration of the Job class is altered so as to take an additional command line argument for the value of K. That is, the command line argument specifies the length of the output of the final Reduce operation.

V. Inverted Index

The inverted index algorithm takes in a text file input and produces an output list which consists of words and the line-numbers on which the words occur. The output is limited to the top 50 words, based on occurrences i.e. the number of lines on which words occur (considering only unique line-numbers).

The implementation is made complex due to the general difficulty of obtaining line-numbers of input files within MrJob Map operations, unless said line-numbers are part of the text body itself. Ultimately, a custom input protocol was written to achieve this functionality.

Within MrJob, *protocols* define the way in which data is packaged and passed to operations. The input protocol defines how data is passed to the first Map operation of a job. Protocols are required to define two methods *read* and *write*, the latter is not relevant to this discussion. The read method is supplied with a line of input data and is required to emit a key-value pair from the input line. For an input protocol, this defines the key-value pairs which are supplied to the first Map operation.

The custom protocol class is based on the default *RawValueProtocol* class, modifying the *read* method to return a line-number of the input data, by making use of a static line-number counter within the custom protocol class. The line counter is incremented on each call made to the *read* method. This approach works because MrJob automatically splits the input data by the newline character by default. Hence, each line sent to the read function is representative of one line of the input text.

The listing below shows the implementation of the custom input protocol.

```
class CustomProtocol(object):
    lineCount=0#static line counter

    def read(self, line):
        decoded=line.decode('utf_8')
        CustomProtocol.lineCount=
            CustomProtocol.lineCount+1
        return (CustomProtocol.
            lineCount),decoded

    def write(self, key, value):
        return '%s\t%s' % (key, value)
```

Fig. 1: Listing of custom input protocol for line-number retrieval.

Once this functionality is implemented, the remainder of the implementation is relatively simple, consisting of three steps. The first phase of 'stop' word removal persists from the previous two algorithms.

The second step involves Map and Reduce operations. The Map operation takes in a delimited string as its value parameter and the line-number of the string as the key parameter. Mappers split their input string into a list of strings and emit key-value pairs wherein the key is the word and the value is the line-number.

The reducer simply takes in the key-values pairs and determines the unique values present within the list of line-numbers for each key, creating a *(word,line-numbers)* tuple. Ultimately, this operation emits a key-value pair wherein the key is the same for all words (NULL), and the value is the tuple mentioned above.

The final step consists of only a Reduce operation which selects the top 50 words based on occurrences (length of line-number list), using a similar approach to the top K words final Reduce operation.

A. Secondary Implementation

A second implementation for the inverted index is proposed: this implementation removes the need for the use of a custom input protocol. The method presupposes the inclusion of line-numbers within the input data as the first element of every line. Thus, this solution theoretically trades generality for performance.

As a result, the overall process does not change drastically, only the Map operation of the first step is adjusted to use the line-number included in the input line as the emitted line-number key.

VI. Results and Discussion

This section presents the results of testing conducted using two input files. A short discussion is also presented, considering these results and the possible improvements which could be made to achieve better performance. The scripts were run on a system with the following specifications:

- CPU: Intel i5 8350u (4C 8T) 3.6 GHz turbo frequency (256 kB L1, 1 MB L2, 6 MB L3) [5].
- RAM: 16 GB 2400 MHz

A. Small Input File

The second implementation of the inverted index algorithm could not be used on this input file as the implementation is reliant on line-numbers being included in the file body, which is not the case with this input file.

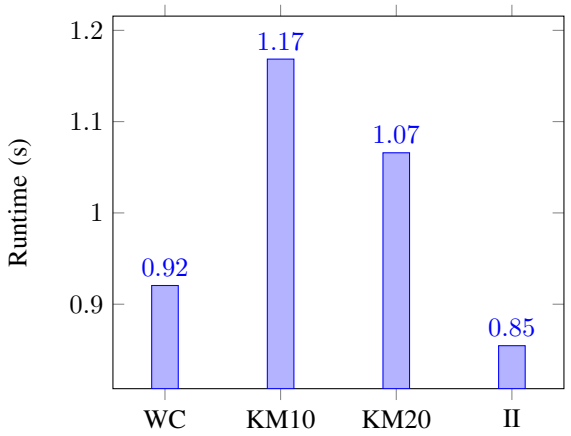

Fig. 2: Performance results using small text file.

The following listings show the results of running the algorithms used with this test input. The results of the word count and inverted index algorithms are truncated for brevity. As stated previously, the inverted index implementation considers only unique line-numbers for its ordering process.

```
"1775"     2
"1776"     4
"1777"     1
"1778"     2
"1779"     1
"1781"     1
"300"      1
"5000"     1
"abundant"          1
"accustomed"        1
"acted"  1
"action"            1
"adams"  3
```

Fig. 3: Truncated results for word count of small file.

```
"Top10" [["congress", 12], ["tories",
    10], ["government", 9], ["state",
    9], ["people", 8], ["british", 7], [
    "constitutions", 7], ["loyalists",
    7], ["men", 7], ["patriot", 7]]

"Top20" [["congress", 12], ["tories",
    10], ["government", 9], ["state",
    9], ["people", 8], ["british", 7], [
    "constitutions", 7], ["loyalists",
    7], ["men", 7], ["patriot", 7], ["
    revolution", 7], ["army", 6], ["
    great", 6], ["jersey", 6], ["
    patriots", 6], ["states", 6], ["
    american", 5], ["leaders", 5], ["
    massachusetts", 5], ["provincial",
    5]]
```

Fig. 4: Results for K most frequent analysis of small file.

```
"Top50" [["congress", "{96, 198, 7, 75,
    77, 175, 145, 17, 179, 81, 88, 155}
    "], ["government", "{128, 90, 9, 42,
    16, 23, 152, 58, 93}"], ["state", "
    {101, 70, 12, 46, 78, 55, 88, 89,
    29}"], ["tories", "{162, 195, 132,
    106, 171, 143, 116, 217, 123}"], ["
    people", "{140, 13, 47, 176, 147,
    148, 151, 158}"], ["constitutions",
    "{38, 102, 12, 46, 19, 55, 29}"], ["
    loyalists", "{166, 167, 143, 116,
    119, 153, 157}"], ["men", "{98, 71,
    44, 109, 178, 216, 187}"], ["patriot
    ", "{138, 108, 15, 181, 182, 215,
    61}"]
```

Fig. 5: Truncated results for inverted index analysis of small file.

B. Large File

This input file makes use of line-numbers at the start of each line, hence the secondary version of the inverted index algorithm can be used here, as indicate below (AII).

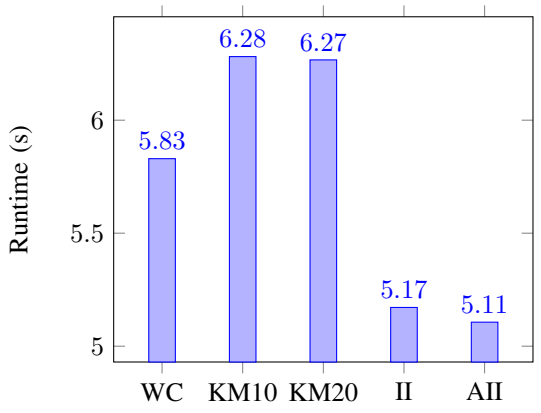

Fig. 6: Performance results using large text file.

It is not practical to list most of the results for this test, due to the size of the output lists. The results for the top K most frequent top 10 and 20 words are shown below, as the size of the output is manageable due to its fixed length.

```
"Top10" [["state", 695], ["good", 672],
    ["man", 610], ["true", 585], ["'",
    526], ["men", 431], ["life", 398], [
    "justice", 359], ["soul", 341], ["
    nature", 336]]

"Top20" [["state", 695], ["good", 672],
    ["man", 610], ["true", 585], ["'",
    526], ["men", 431], ["life", 398], [
    "justice", 359], ["soul", 341], ["
    nature", 336], ["plato", 326], ["
    knowledge", 292], ["replied", 273],
    ["truth", 263], ["great", 258], ["
    things", 256], ["evil", 233], ["mind
    ", 229], ["time", 217], ["reason",
    200]]
```

Fig. 7: Results of K most frequent analysis of large text file.

C. Discussion

Based on the performance results obtained when using the large text file as an input, it does not appear that there is a significant difference between the two approaches proposed for the inverted index algorithm. As a result, the first approach may be preferred for the generality of its application. That said, it is unclear whether or not it is safe to use this approach when running on a cluster. Moreover, the approach used is not compatible with multiple input files due to the implementation of the custom protocol described earlier. That is, there is seemingly no way in which to reset the line count when a new file begins.

Another point of discussion is the use of a standalone step for the purpose of removing stop words. It is thought that this approach will have minimal effect on performance, considering there is no devolution to a single reducer and that the removal of stop words would have to be done at some stage in any case.

It is interesting to note the large difference in performance between the top K algorithm and the two inverted index approaches. Both of these approaches devolve to a single reducer in their final step, and both approaches make use of the same built-in sorter and selector. Despite this, the inverted index algorithms perform better in the testing described earlier.

It was identified in selective testing that the use of an intermediate Combine operation hindered the performance of both the word count algorithm and the top K most frequent algorithm, as shown in the adjacent graph (-C indicates the use of a Combine operation). This is particularly interesting considering that most tutorials suggest the use of the Combine operation.

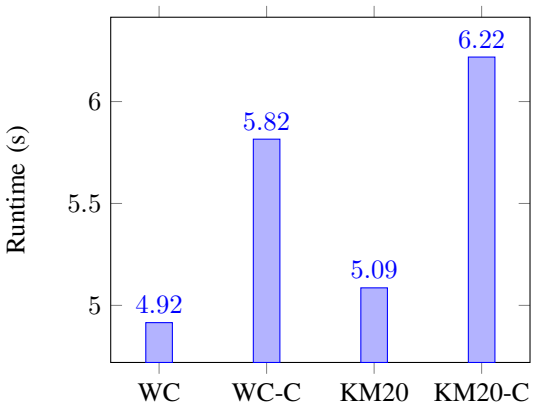

Fig. 8: Performance comparison considering effect of Combine step.

VII. Conclusion

A simple explanation of the MapReduce programming model has been presented. This explanation was informed by a practical implementation of MapReduce solutions, using MrJob in Python. The implemented algorithms achieve their respective goals, but could be improved in future work through reducing the number of steps involved; simplifying multi-step operations to a single step. Additionally, a more complex analysis of the use of a Combine operation should be attempted to evaluate its efficiency, and the input size threshold after which it yields superior performance.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, 2004, pp. 137–150.

[2] *Apache hadoop*, https://hadoop.apache.org/, (Accessed on 13/04/2019),

[3] *Github - kozyraki/phoenix: An api and runtime environment for data processing with mapreduce for shared-memory multi-core & multiprocessor systems.* https://github.com/kozyraki/phoenix, (Accessed on 13/04/2019),

[4] *Mrjob fffdfffdfffd mrjob v0.5.10 documentation*, https://pythonhosted.org/mrjob/, (Accessed on 13/04/2019),

[5] *Intel core i5-8350u processor (6m cache, up to 3.60 ghz) product specifications*, https://ark.intel.com/content/www/us/en/ark/products/124969/intel-core-i5-8350u-processor-6m-cache-up-to-3-60-ghz.html, (Accessed on 15/03/2019),

PSEUDOCODE

WORDCOUNT

Algorithm 1 WordCount

```
1:  procedure STOPWORDMAP(key,value)
2:      lineNo ← line number - Key
3:      line ← input line - Value
4:      wordList=line.split()
5:      for each word in wordList do
6:          if word.lower() not in StopWords then
7:              emit Tuple (key=lineNo, value=word)
8:  procedure STOPWORDREDUCE(key,values)
9:      lineNo ← line number - Key
10:     words ← word - Values
11:     wordList=toDelimitedList(words)
12:     emit Tuple (key=lineNo, value=wordList)
13: procedure WORDCOUNTMAP(key,value)
14:     lineNo ← line number - Key
15:     wordList ← word - Value
16:     words=wordList.split()
17:     for each word in words do
18:         emit Tuple (key=word, value=1)
19: procedure WORDCOUNTCOMBINE(key,value)
20:     word ← word - Key
21:     ones ← ones - Value
22:     totalCount=sum(ones)
23:     emit Tuple (key=word, value=totalCount)
24: procedure WORDCOUNTREDUCE(key,values)
25:     word ← word - Key
26:     counts ← counts - Values
27:     totalCount=sum(counts)
28:     emit Tuple (key=word, value=totalCount)
```

Algorithm 2 KMostFrequent

```
1: procedure STOPWORDMAP(key,value)
2:     lineNo ← line number - Key
3:     line ← input line - Value
4:     wordList=line.split()
5:     for each word in wordList do
6:         if word.lower() not in StopWords then
7:             emit Tuple (key=lineNo, value=word)
8: procedure STOPWORDREDUCE(key,values)
9:     lineNo ← line number - Key
10:    words ← word - Values
11:    wordList=toDelimitedList(words)
12:    emit Tuple (key=lineNo, value=wordList)
13: procedure WORDCOUNTMAP(key,value)
14:    lineNo ← line number - Key
15:    wordList ← word - Value
16:    words=wordList.split()
17:    for each word in words do
18:        emit Tuple (key=word, value=1)
19: procedure WORDCOUNTCOMBINE(key,value)
20:    word ← word - Key
21:    ones ← ones - Value
22:    totalCount=sum(ones)
23:    emit Tuple (key=word, value=totalCount)
24: procedure WORDCOUNTREDUCE(key,values)
25:    word ← word - Key
26:    counts ← counts - Values
27:    totalCount=sum(counts)
28:    emit Tuple (key=None, value= Tuple(key=word,value=totalCount))
29: procedure KMOSTREDUCE(key,values)
30:    nullKey ← nullKey - Key
31:    tuples ← word-count-tuples - Values
32:    mostFrequent = KMost(tuples).byWordCount()
33:    emit Tuple (key='TOP K', value=MostFrequent)
```

Algorithm 3 InvertedIndex

```
1: procedure StopWordMap(key,value)
2:     lineNo ← line number - Key
3:     line ← input line - Value
4:     wordList=line.split()
5:     for each word in wordList do
6:         if word.lower() not in StopWords then
7:             emit Tuple (key=lineNo, value=word)
8: procedure StopWordReduce(key,values)
9:     lineNo ← line number - Key
10:    words ← word - Values
11:    wordList=toDelimitedList(words)
12:    emit Tuple (key=lineNo, value=wordList)
13: procedure LineNumMap(key,value)
14:    lineNo ← line number - Key
15:    wordList ← word - Value
16:    words=wordList.split()
17:    for each word in words do
18:        emit Tuple (key=word, value=lineNo)
19: procedure LineNumReduce(key,values)
20:    word ← word - Key
21:    lines ← lines - Values
22:    uniqueLines=Unique(lines)
23:    emit Tuple (key=None, value= Tuple(key=word, value= toDelimitedString(uniqueLines)))
24: procedure Top50Reduce(key,values)
25:    None ← None - Key
26:    tuples ← tuples - Values
27:    mostFrequent = 50Most(tuples).byOccurrences()
28:    emit Tuple (key='TOP 50',mostFrequent)
```

Algorithm 4 InvertedIndex

```
1: procedure STOPWORDMAP(key,value)
2:     key ← Null - Key
3:     line ← input line - Value
4:     wordList=line.split()
5:     for each word in wordList do
6:         if word.lower() not in StopWords then
7:             lineNo=wordList.first().toInt()
8:             emit Tuple (key=lineNo, value=word)
9: procedure STOPWORDREDUCE(key,values)
10:    lineNo ← line number - Key
11:    words ← word - Values
12:    wordList=toDelimitedList(words)
13:    emit Tuple (key=lineNo, value=wordList)
14: procedure LINENUMMAP(key,value)
15:    lineNo ← line number - Key
16:    wordList ← word - Value
17:    words=wordList.split()
18:    for each word in words do
19:        emit Tuple (key=word, value=lineNo)
20: procedure LINENUMREDUCE(key,values)
21:    word ← word - Key
22:    lines ← lines - Values
23:    uniqueLines=Unique(lines)
24:    emit Tuple (key=None, value= Tuple(key=word, value= toDelimitedString(uniqueLines)))
25: procedure TOP50REDUCE(key,values)
26:    None ← None - Key
27:    tuples ← tuples - Values
28:    mostFrequent = 50Most(tuples).byOccurrences()
29:    emit Tuple (key='TOP 50',mostFrequent)
```

Stop Words

a , about , above , after , again , against , ain , all , am, an , and , any , are , aren , aren ' t , as , at , be , because , been , before , being , below , between , both , but , by , can , couldn , couldn ' t , d , did , didn , didn ' t , do , does , doesn , doesn ' t , doing , don , don ' t , down , during , each , few , for , from , further , had , hadn , hadn ' t , has , hasn , hasn ' t , have , haven , haven ' t , having , he , her , here , hers , herself , him , himself , his , how , i , if , in , into , is , isn , isn ' t , it , it ' s , its , itself , just , ll , m, ma, me, mightn , mightn ' t , more , most , mustn , mustn ' t , my, myself , needn , needn ' t , no , nor , not , now , o , of , off , on , once , only , or , other , our , ours , ourselves , out , over , own , re , s , same , shan , shan ' t , she , she ' s , should , should ' ve , shouldn , shouldn ' t , so , some , such , t , than , that , that ' ll , the , their , theirs , them , themselves , then , there , these , they , this , those , through , to , too , under , until , up , ve , very , was , wasn , wasn ' t , we , were , weren , weren ' t , what , when , where , which , while , who , whom, why , will , with , won, won ' t , wouldn , wouldn ' t , y , you , you ' d , you ' ll , you ' re , you ' ve , your , yours , yourself , yourselves , could , he ' d , he ' ll , he ' s , here ' s , how ' s , i ' d , i ' ll , i ' m, i ' ve , let ' s , ought , she ' d , she ' ll , that ' s , there ' s , they ' d , they ' ll , they ' re , they ' ve , we ' d , we ' ll , we ' re , we ' ve , what ' s , when ' s , where ' s , who ' s , why ' s , would , able , abst , accordance , according , accordingly , across , act , actually , added , adj , affected , affecting , affects , afterwards , ah , almost , alone , along , already , also , although , always , among , amongst , announce , another , anybody , anyhow , anymore , anyone , anything , anyway , anyways , anywhere , apparently , approximately , arent , arise , around , aside , ask , asking , auth , available , away , awfully , b , back , became , become , becomes , becoming , beforehand , begin , beginning , beginnings , begins , behind , believe , beside , besides , beyond , biol , brief , briefly , c , ca , came , cannot , can ' t , cause , causes , certain , certainly , co , com , come , comes , contain , containing , contains , couldnt , date , different , done , downwards , due , e , ed , edu , effect , eg , eight , eighty , either , else , elsewhere , end , ending , enough , especially , et , etc , even , ever , every , everybody , everyone , everything , everywhere , ex , except , f , far , ff , fifth , first , five , fix , followed , following , follows , former , formerly , forth , found , four , furthermore , g , gave , get , gets , getting , give , given , gives , giving , go , goes , gone , got , gotten , h , happens , hardly , hed , hence , hereafter , hereby , herein , heres , hereupon , hes , hi , hid , hither , home , howbeit , however , hundred , id , ie , im , immediate , immediately , importance , important , inc , indeed , index , information , instead , invention , inward , itd , it ' ll , j , k , keep , keeps , kept , kg , km, know , known , knows , l , largely , last , lately , later , latter , latterly , least , less , lest , let , lets , like , liked , likely , line , little , ' ll , look , looking , looks , ltd , made , mainly , make , makes , many , may , maybe , mean , means , meantime , meanwhile , merely , mg, might , million , miss , ml , moreover , mostly , mr , mrs , much , mug , must , n , na , name , namely , nay , nd , near , nearly , necessarily , necessary , need , needs , neither , never , nevertheless , new , next , nine , ninety , nobody , non , none , nonetheless , noone , normally , nos , noted , nothing , nowhere , obtain , obtained , obviously , often , oh , ok , okay , old , omitted , one , ones , onto , ord , others , otherwise , outside , overall , owing , p , page , pages , part , particular , particularly , past , per , perhaps , placed , please , plus , poorly , possible , possibly , potentially , pp , predominantly , present , previously , primarily , probably , promptly , proud , provides , put , q , que , quickly , quite , qv , r , ran , rather , rd , readily , really , recent , recently , ref , refs , regarding , regardless , regards , related , relatively , research , respectively , resulted , resulting , results , right , run , said , saw , say , saying , says , sec , section , see , seeing , seem , seemed , seeming , seems , seen , self , selves , sent , seven , several , shall , shed , shes , show , showed , shown , showns , shows , significant , significantly , similar , similarly , since , six , slightly , somebody , somehow , someone , somethan , something , sometime , sometimes , somewhat , somewhere , soon , sorry , specifically , specified , specify , specifying , still , stop , strongly , sub , substantially , successfully , sufficiently , suggest , sup , sure , take , taken , taking , tell , tends , th , thank , thanks , thanx , thats , that ' ve , thence , thereafter , thereby , thered , therefore , therein , there ' ll , thereof , therere , theres , thereto , thereupon , there ' ve , theyd , theyre , think , thou , though , thoughh , thousand , throug , throughout , thru , thus , til , tip , together , took , toward , towards , tried , tries , truly , try , trying , ts , twice , two , u , un , unfortunately , unless , unlike , unlikely , unto , upon , ups , us , use , used , useful , usefully , usefulness , uses , using , usually , v , value , various , ' ve , via , viz , vol , vols , vs , w, want , wants , wasnt , way , wed , welcome , went , werent , whatever , what ' ll , whats , whence , whenever , whereafter , whereas , whereby , wherein , wheres , whereupon , wherever , whether , whim , whither , whod , whoever , whole , who ' ll , whomever , whos , whose , widely ,

willing , wish , within , without , wont , words , world , wouldnt , www, x , yes , yet , youd , youre , z , zero , a ’ s , ain ’ t , allow , allows , apart , appear , appreciate , appropriate , associated , best , better , c ’ mon , c ’ s , cant , changes , clearly , concerning , consequently , consider , considering , corresponding , course , currently , definitely , described , despite , entirely , exactly , example , going , greetings , hello , help , hopefully , ignored , inasmuch , indicate , indicated , indicates , inner , insofar , it ’ d , keep , keeps , novel , presumably , reasonably , second , secondly , sensible , serious , seriously , sure , t ’ s , third , thorough , thoroughly , three , well , wonder