MACQUARIE UNIVERSITY

Faculty of Science and Engineering

Department of Computing

COMP3210 - BIG DATA (SEMESTER 1)

ASSIGNMENT - 2

Task-1 (15%)

Task-2 (15%)

Task-3 (35%)

Task-4 (35%)

Student Name : KODALI SAHITHI

Student ID : 45712050

Practical : Thursday, 10am - 12pm

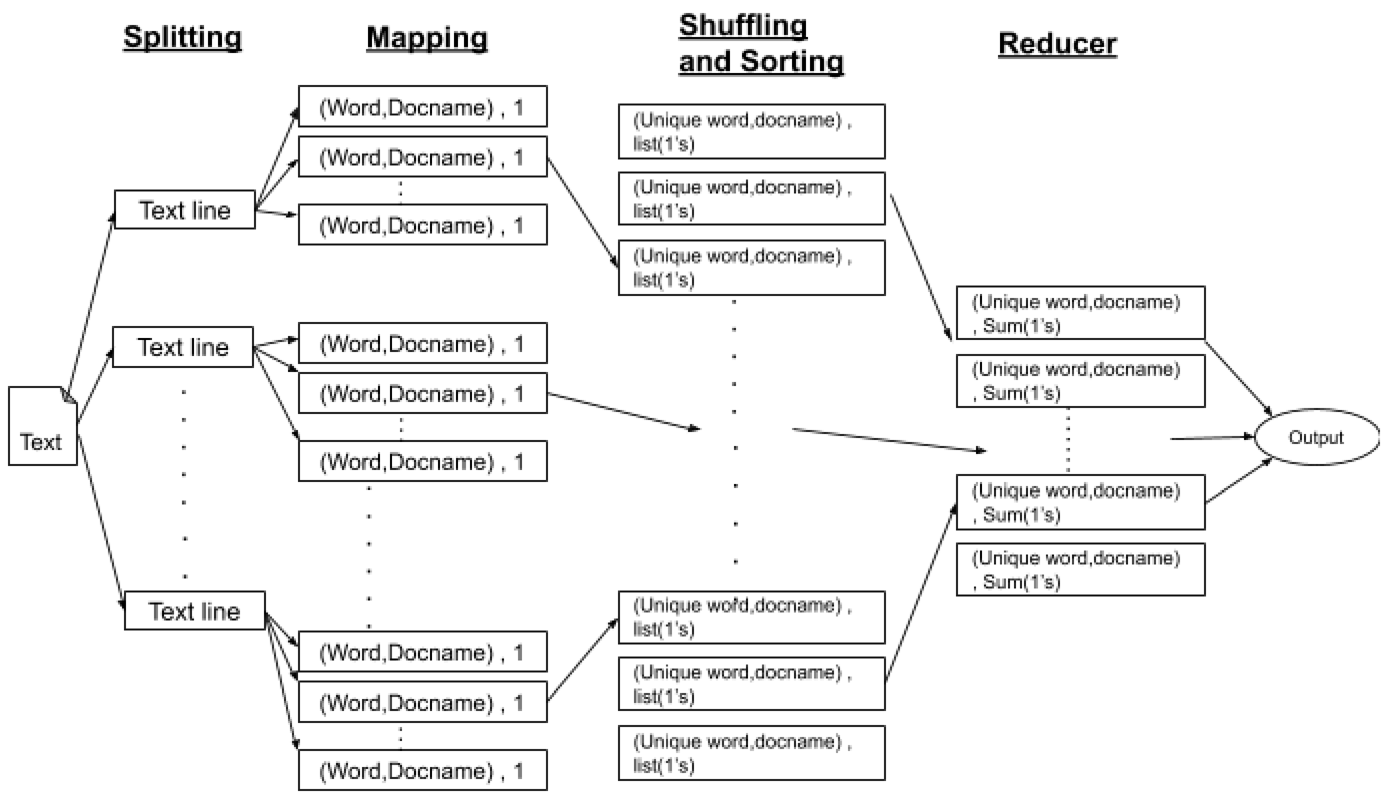
Contact Details : 0451116411

Date : 10-05-2020

Task - 1 : Calculate the count of number of occurrences of each word in the text of Tweets.

FLOWCHART:

The flowchart below takes each doc in text and splits it into each word with the corresponding docname. Each word is mapped to the number 1 as value. The output from mapper is shuffled and sorted to get a list of 1's for each unique word. This is sent to the reducer where the sum of the list of 1's gives the count of each word as output.



PSEUDOCODE:

Mapper :

Mapper takes the content of the file with DocID and Text as two columns with the text processed and cleaned. Each doc is split into words. The first word is considered as docname and for every word in the remaining words the key is produced as the word and docname. Each key i.e word is mapped with number 1 as value and produced as Output.

Input : (docName , Content_of_file)
Function:

```
For each line in the Input:
    words = line.split(" ")        #Split line with delimiter(" ") to get the content of Doc
    doc_name = words [0]           #Assigning first word to docname
    for word in words[1: ]:        #For the remaining words of text
        Key = word + ',' + doc_name   #Add key as word, docname
        Yield Output
```

Output : ((word, doc_Name) , 1)

Reducer :

Reducer takes the input from Mapper which is a list of 1's as values for each key i.e word. For each word i.e key sum of the 1's is evaluated as value, produced as output.

Input : ((word, doc_Name) , 1)

Function:

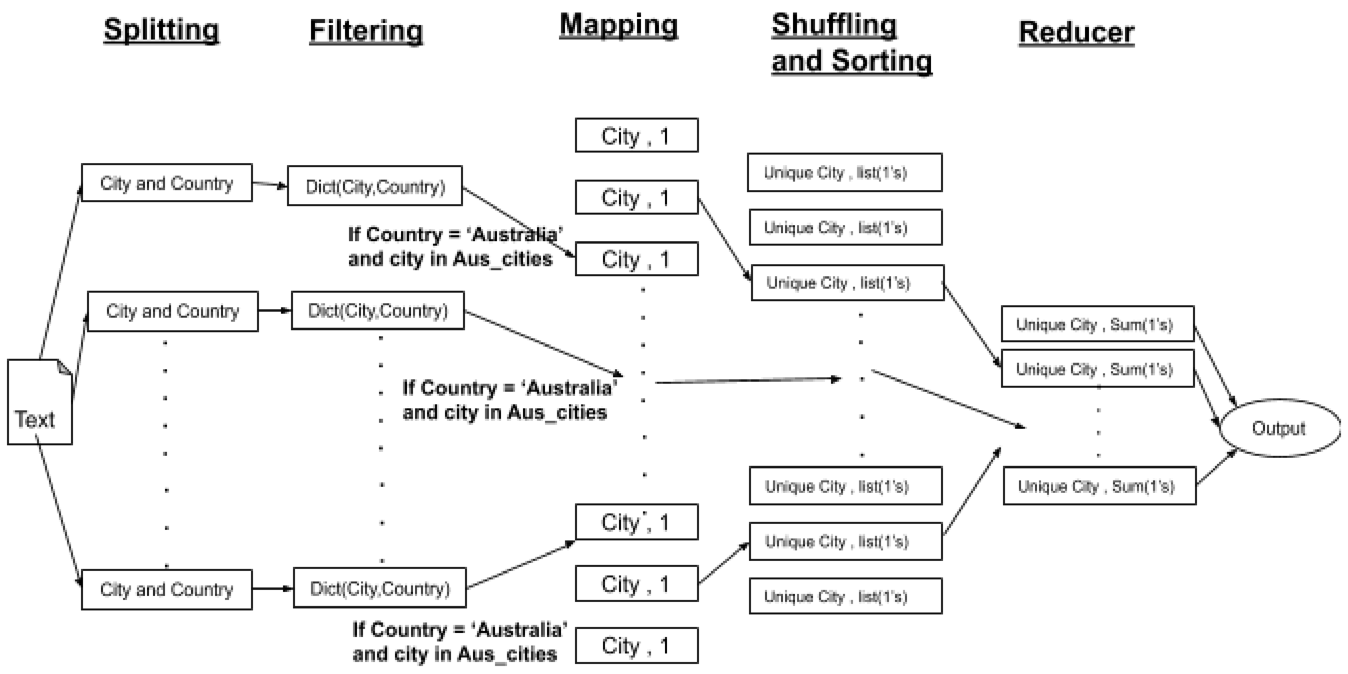
```
For each line in the value of Input:
    Count = sum(value)     #Sums the value i.e list of 1's of each unique word/key
    Yield Output       #yields the sum as ((word,doc_name), count) key,value pair
```

Output : ((word, doc_Name) , count)

Task - 2 : Calculate the count of number of tweets for a list of different cities in Australia.

FLOWCHART:

The flowchart below takes each doc in text and splits it into city and country. Then the city and country are put into a dictionary to pair each city with the respective country value. Then each city is sent to the mapper if the city is in aus_cities and the respective value has the country as 'Australia' which below is written in Filtering. The mapper maps each city to value 1 as count and its output is shuffled and sorted and sent a value of list of 1's for every unique key before sending to the Reducer. The reducer now sums up the value to give the count of each city.



PSEUDOCODE:

Mapper:

Mapper takes the input file and its contents. The function divides each line into city and country pairs, adding them to a dictionary. Check if the city is in 'Aus_cities' and if the corresponding country i.e value is 'australia', if so add the corresponding key which is the city and output (city,1).

```
Input :    (docName , Content_of_file)
Function :
For each line in the Input:
    Words = line.split("\t")       #Split line with delimiter('\t') to obtain city and country as words
  Pairs = { }       #Create a Dictionary and match each city to its Country(key = city & value=country)
  Pairs[words[0]] = words[1: ]      #Add city and corresponding country as key and value in dict
  If city in Aus_cities:                 #if city is  a valid Australian city
    If 'Australia' in pairs[words[0]]:   #If the city value  is Australia then get the corresponding key i.e city
       Yield output               #Corresponding city as key and 1 as value
Output : (City,1)
```

Reducer:

Reducer takes in the input from Mapper i.e words which are cities and adds the values which are list of 1's and produces the key value pair (Word, sum(1's)) i.e count of cities.

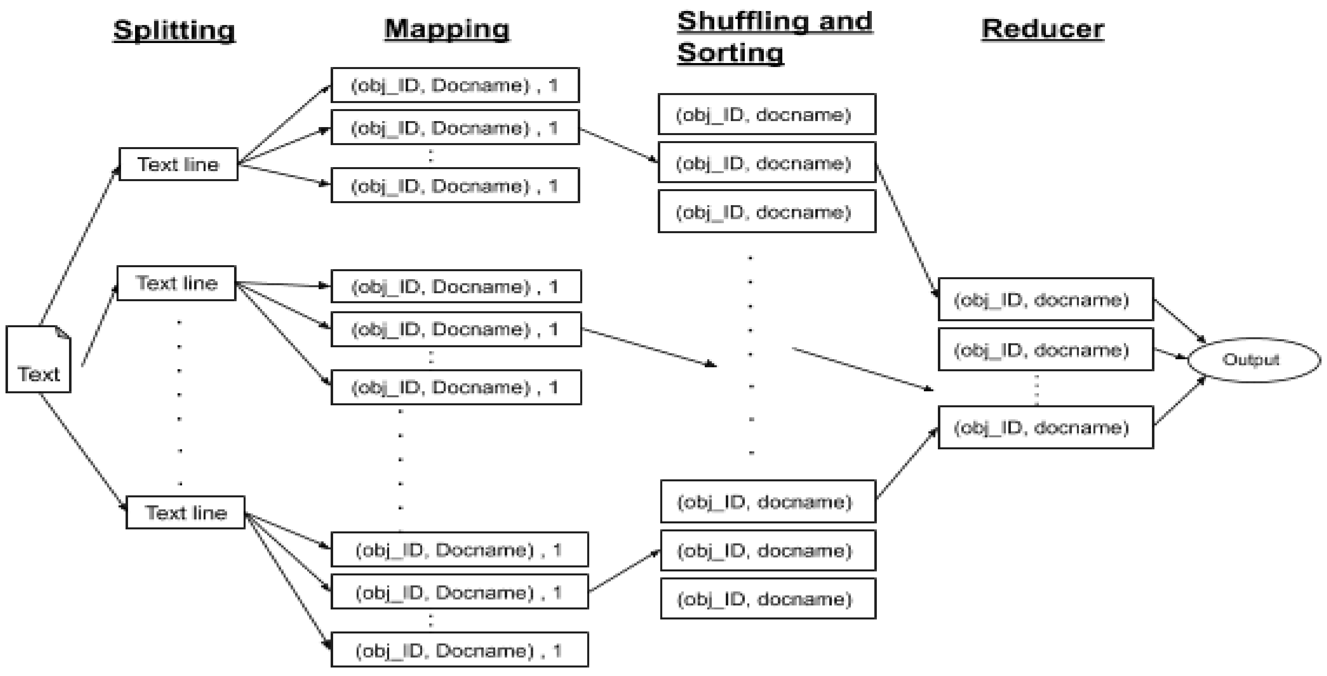
```
Input :   (word , list of 1's)          #word is a city
Function :
For each line in the input:     #Value is list of 1's
          Yield Output       #Produces the sum of 1's of each word(city)

Output : (word, sum(1's) )
```

Task - 3 :

A.) Implement the Merge Sort algorithm using Map-Reduce.

FLOWCHART:

The flowchart below takes each doc in text and splits it into obj_ID and docname and maps each ID to 1 as value. Then each mapper output is shuffled and sorted before sent to the Reducer. The reducer implements Merge Sort Algorithm which is an inbuilt function of Reducer and sorts the obj_ID's.



PSEUDOCODE:

Mapper:

Mapper takes the content of the file whose each line is split into words. The file here as Docname and obj_ID as two columns with the text processed and cleaned. The first word is considered as docname and second word as obj_ID. Each key as (obj_ID,docname) is mapped with number 1 as value and produced as Output.

Input : (docName , Content_of_file)

Function:

```
For each line in the Input:
   words = line.split("\t ")        #Split line with delimiter("\t") to get the contents of each Doc
   doc_name = words [0]       #Assigning first word to docname
   obj_ID = words [1]         #Assigning next word to obj_ID
   ID = obj_ID+ ',' + doc_name   #Add key as obj_ID, docname
   Yield Output
```

Output : ((obj_ID, doc_Name) , 1)

Reducer:

Reducer takes the input from Mapper which is a list of 1's as values for each key i.e obj_ID. All the ID's are now sorted based on Merge sort Algorithm which is an inbuilt algorithm of a Reducer. Since the reducer implements Merge sort based on the keys, we need to ensure to send the ID as key only.

Input : ((obj_ID, doc_Name) , 1)

Function:

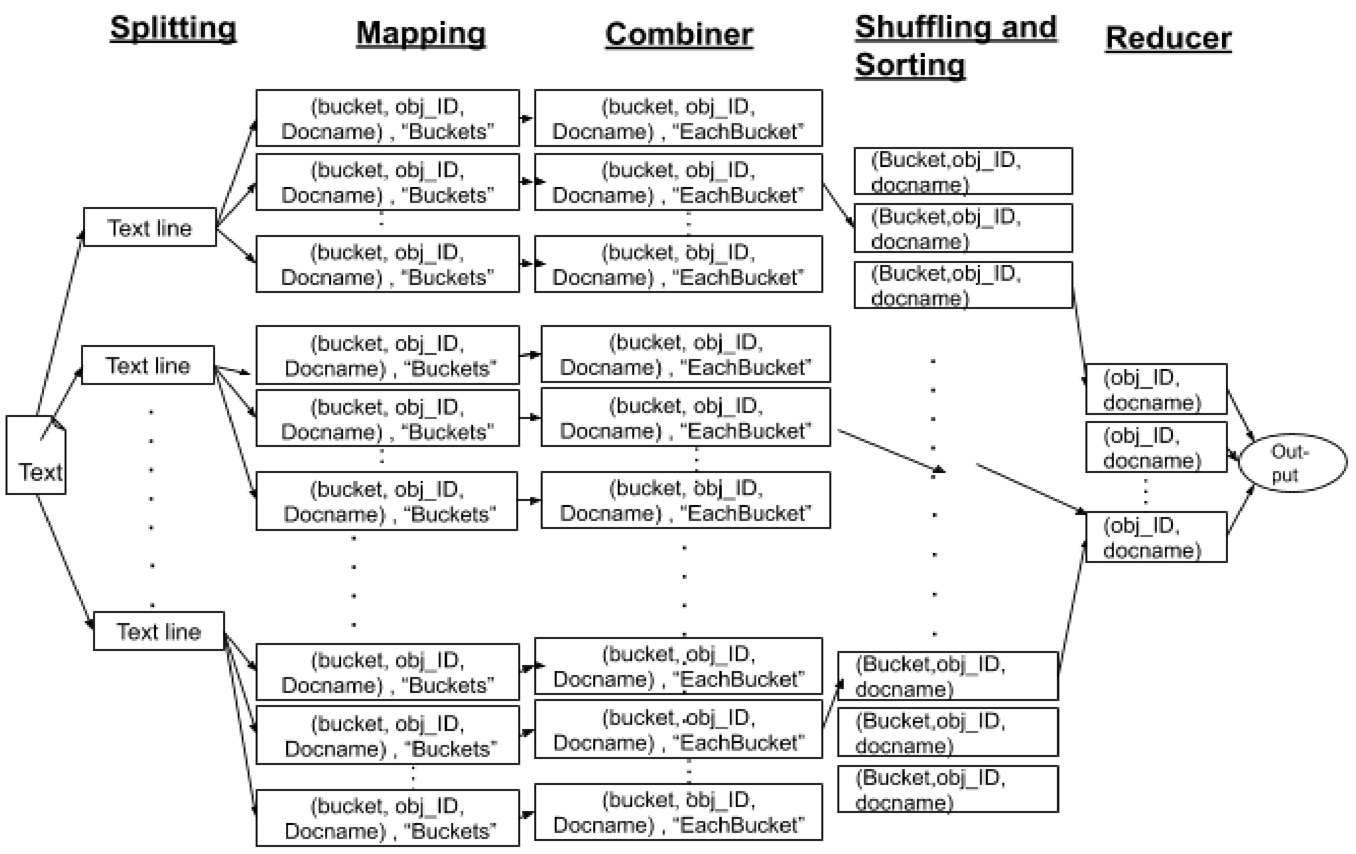
```
For each line in the value of Input:
    (ID,docname) = ID_docname.split(",")    #Split the key with delimiter(",")  into obj_ID and docname
    Yield Output       #yields the output with sorted keys which are the obj_ID's here
```

Output : (obj_ID, doc_Name)

B.) Implement the Bucket Sort algorithm using Map-Reduce.

FLOWCHART:

The flowchart below takes each doc in text and splits it into obj_ID and docname. Each obj_ID is multiplied with the size of the text file to get buckets for each obj_ID. Now each bucket along with ID, docname is sent as a key to give out as output. The Combiner takes each bucket and sorts the obj_ID with respect to the bucket. Then each bucket output is shuffled and sorted before sent to the Reducer to get the obj_ID and docname in sorted order.



PSEUDOCODE:

Mapper:

Mapper takes the content of the file whose each line is split into words. The file here as Docname and obj_ID as two columns with the text processed and cleaned. The first word is considered as docname and second word as obj_ID. Each obj_ID is multiplied with the total number of documents(n) in the text file to get a bucket. Now each bucket along with obj_id and docname is mapped as buckets and sent as output. Each bucket for each obj_ID is now sorted inside the Buckets using Combiner. Then the output is shuffled and sorted based on Bucket numbers from Combiner and sorted to give the obj_ID, docname in sorted order. (In the second Reducer *'merge Sort'* is implemented on the Buckets)

Input : (docName , Content_of_file)

Function:

```
For each line in the Input:
    words = line.split("\t ")        #Split line with delimiter("\t") to get the contents of each Doc
    doc_name = words [0]       #Assigning first word to docname
    obj_ID = words [1]       #Assigning next word to obj_ID
    Bucket = n* float(obj_ID)  # Each bucket is obtained by multiplying no of docs to obj_ID
    Yield Output
```

Output : ((Bucket, obj_ID, doc_Name) , "Buckets")

Combiner:

Combiner takes the input from Mapper which is Bucket, obj_ID, docname and value. The key is split to obtain the buckets , obj_id an doc name. Then each bucket and obj_ID and docname are sent as a key to get each bucket values/ID's sorted in their respective buckets based on bucket value and outputs 'eachbucket' sorted form.

Input : ((Bucket, obj_ID, doc_Name) , "Buckets")

Function:

```
For each line in the value of Input:
    (bucket, obj_id) = key[0], key[1]     #Obtain the bucket and ID with respective list indexes
    Docname = key[2]                      #Obtain the docname
    Yield Output                  #Yield Output for each bucket
```

Output : ((Bucket, obj_ID, doc_Name), "Each bucket")

Reducer:

Reducer takes the input from Combiner and sorts the keys based on the value of Buckets. This is basically a Merge Sort on the Buckets obtaine dsince Reducer has the in-built Merge sort Algorithm implemented. Thus, all the buckets are obtained in a sorted manner. Extract the respective obj_ID and docname as Key and value of the Output.

Input : ((Bucket, obj_ID, doc_Name), "Each bucket")

Function:

```
For each line in the value of Input:
    (bucket, obj_id) = key[0], key[1]     #Obtain the bucket and ID with respective list indexes
    Docname = key[2]                      #Obtain the docname
    Yield Output                  # yield output for all the buckets
```

Output : (obj_ID, doc_Name)

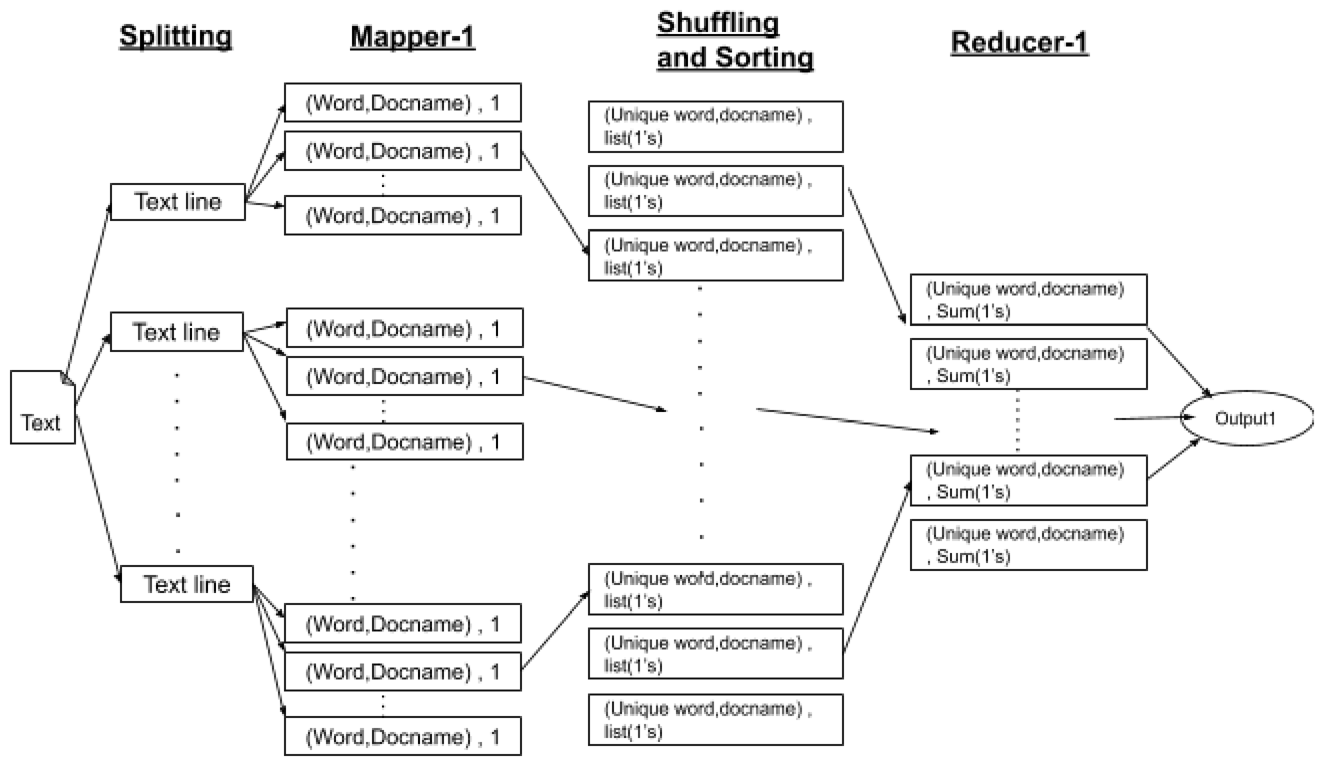
NOTE: Both the Merge sort and Bucket sort can be implemented in a single Map-reduce by using one reducer to divide the ID into buckets and the other to Merge sort the buckets since it is the in-built algorithm of a Reducer. However, for detailed explanation of both the algorithms two different implementations with flowchart and pseudo code have been written.
(In this case, 1 mapper and 2 Reducers).

Task - 4 : Implement the TF-IDF algorithm using Map-Reduce for the term "health" in the text of the Tweets

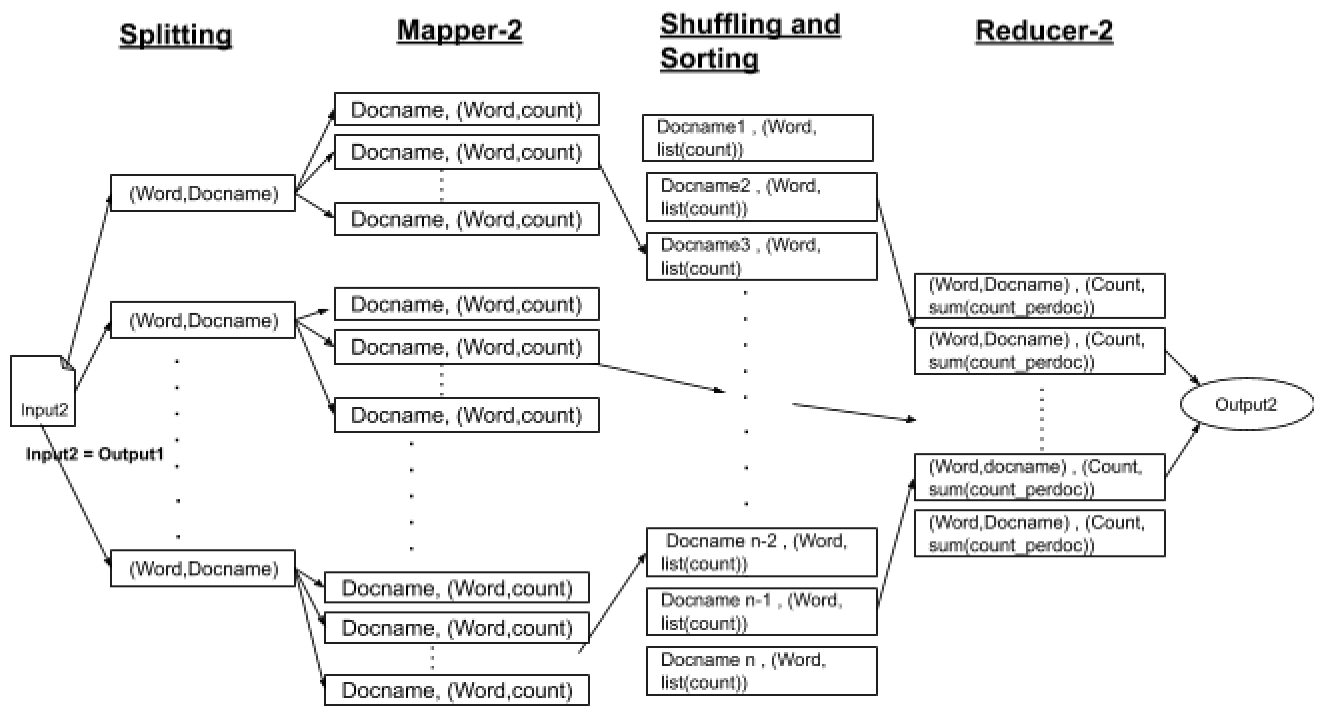
FLOWCHART:

The tf-idf algorithm Implementation can be done using 3 map reduce jobs. However, for ease of understanding and coding this is implemented in 4 Map- reduce jobs with the last job just using a Mapper to calculate tf-idf.

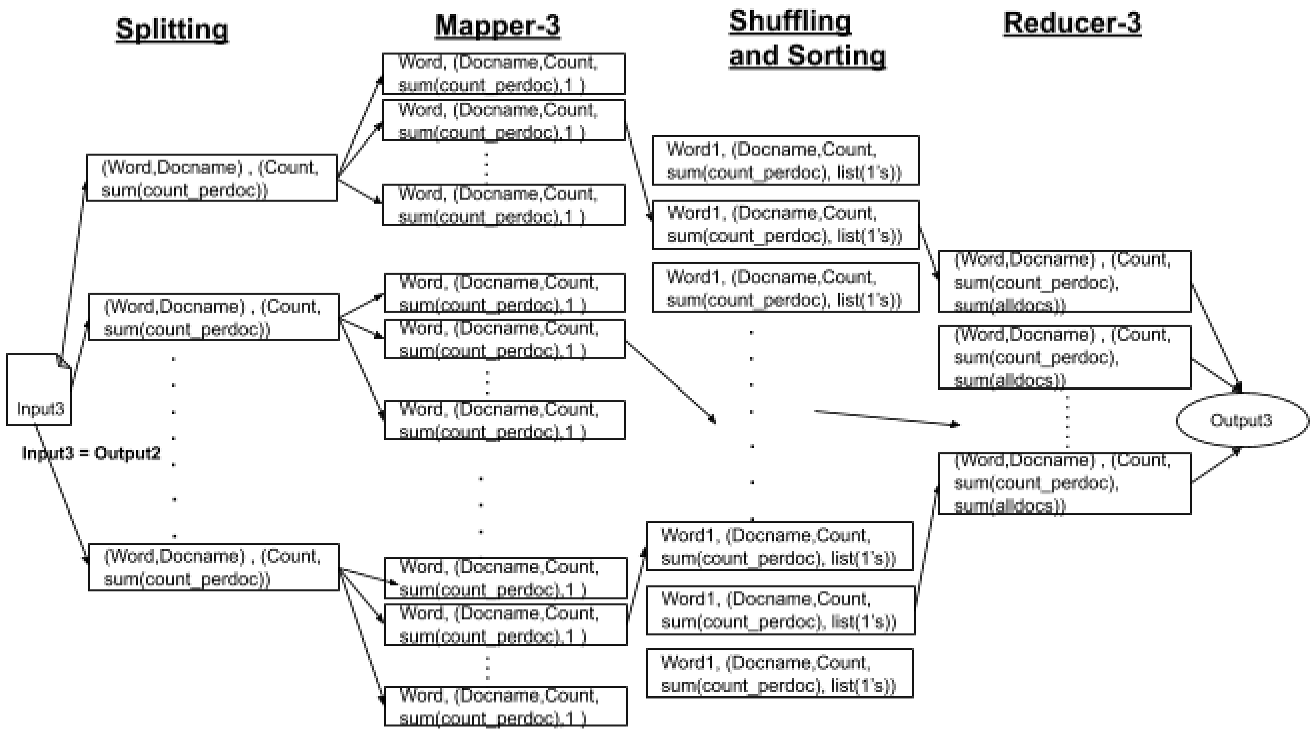
Map Reduce -1: (word frequency in Doc)



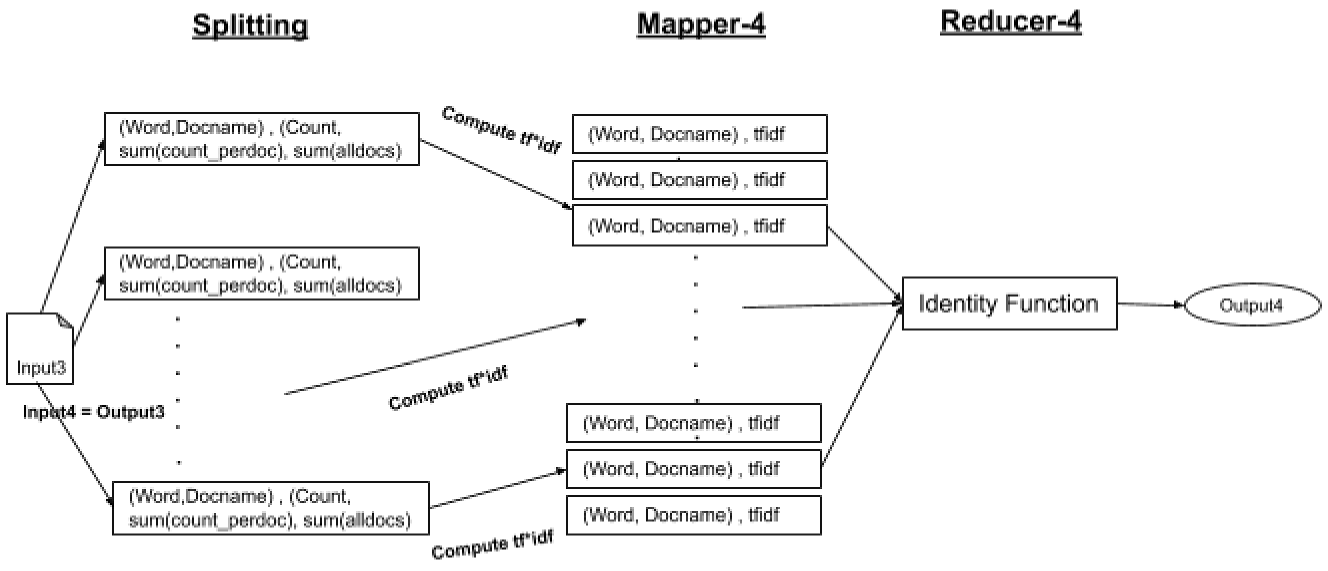
Map Reduce - 2: (Word count for Docs)


Splitting
Mapper-2
Shuffling and Sorting
Reducer-2
Input2
Input2 = Output1
(Word,Docname)
Docname, (Word,count)
Docname1 , (Word, list(count))
Docname2 , (Word, list(count))
Docname3 , (Word, list(count)
Docname n-2 , (Word, list(count))
Docname n-1 , (Word, list(count))
Docname n , (Word, list(count))
(Word,Docname) , (Count, sum(count_perdoc))
(Word,docname) , (Count, sum(count_perdoc))
Output2

Map Reduce - 3: (Word frequency in Corpus(All docs))


Splitting
Mapper-3
Shuffling and Sorting
Reducer-3
Input3
Input3 = Output2
(Word,Docname) , (Count, sum(count_perdoc))
Word, (Docname,Count, sum(count_perdoc),1)
Word1, (Docname,Count, sum(count_perdoc), list(1's))
(Word,Docname) , (Count, sum(count_perdoc), sum(alldocs))
Output3

Map Reduce - 4: (Computer TF-IDF)



PSEUDOCODE:

The TF-IDF algorithm implemented below makes use of 4 Map Reduce Jobs. However, it can also be executed with 3 Map Reduce Jobs. For convenience in coding, this implementation has been splitted into 4 Map Reduce jobs which are detailed as below:

MAP REDUCE - 1 (Word frequency in Doc)

Mapper 1:

The first Mapper takes the content of the file whose each lien is split into words. The file here as DocID and Text as two columns with the text processed and cleaned. The first word is considered as docname and for every word in the remaining words the key is produced as the word and docname. Each key i.e word is mapped with number 1 as value and produced as Output.

Input : (docName , Content_of_file)

Function:

```
For each line in the Input:
    words = line.split(“ “)          #Split line with delimiter(“ ”) to get the content of Doc
    doc_name = words [0]             #Assigning first word to docname
    for word in words[1: ]:          #For the remaining words of text
        Key = word + ‘,’ + doc_name  #Add key as word, docname
        Yield Output
```

Output : ((word, doc_Name) , 1)

Reducer 1:

This Reducer takes the input from Mapper 1 which is a list of 1’s as values for each key i.e word. For each word i.e key sum of the 1’s is evaluated as value, produced as output.

Input : ((word, doc_Name) , 1)

Function:

```
For each line in the value of Input:
     Count = sum(value)     #Sums the value i.e list of 1’s of each unique word/key
     Yield Output           #yields the sum as ((word,doc_name), count) key,value pair
```

Output : ((word, doc_Name) , count)

MAP REDUCE - 2 (Word count for docs)

Mapper 2:

The second Mapper overrides the first Mapper function whose input is from the Reducer function before. This Mapper takes the word,doc a skey and count as value. The key is split to give docname as key for the reducer, as the aim is to find the word count in doc and providing docname as key allows reducer to sort based on the key. The Mapper outputs docname as key and word,its count as the value to the reducer.

Input : ((word, doc_Name) , count)

Function:

```
For each line in the Input:
    word = word.split(“,”)      #Split line with delimiter(“, ”) to obtain key,value (word,docname)
    word_freq = word[0] + count     #Concatenate the word and its count to get the value
    Yield Output      #yields key,value as (doc_name,word_freq) where word_freq is (word, count)
```

Output : (doc_Name , (word, count))

Reducer 2:

The Reducer takes the Input from the Mapper2 which is sorted based on the key docname. The value is split to obtain word and its count. Each word and count in each line are added to a new list. As a new word is found in the same doc, the count is added and each doc word count is produced by the reducer with key as (docname,word),(count of that word, word count of doc)

Input : (doc_Name , (word, count))

Function:

```
Words = [ ]
Freq = [ ]
Wordcount_perdoc = 0
For each word in the value of Input:
    (word,count) = word.split(“,”)     #Split line with delimiter(“, ”) to obtain key,value (word, count)
    words.append(word)               #Append each word into the word list
    freq.append(count)               #Append each count into the freq list
    Wordcount_perdoc += count   #sum the count to get the number of words in each doc
    Yield output                  #yields output as (word, docname),(count,words_indoc) as key,val pair
```

Output : ((word,docName) , (count , N))

MAP REDUCE - 3 (Word frequency in Corpus(All docs))

Mapper 3:

The Mapper 3 overrides the Mapper 2 and takes Input from Reducer 2. It splits the key to get docname and word. Splits the value to get count of word and count of words in Doc. The output is yielded as word as key and (docname,Count of word, Count of words in doc, 1)

Input : ((word,docName) , (count, N))

Function:

```
For each line in the Input:
  (word, doc_name) = key[0], key[1]  #Get each word in key to get the word and corresponding docname
  (count,N) = value[0], value[1]      #Get each word in value to get word count and word count per doc(N)
  Yield Output                        #yield key, value pair as word,(docname,count, wordcount_perdoc ,1)
```

Output : (word, (docName, count, N, 1))

Reducer 3:

The Reducer 3 takes the Input from Mapper 3 and splits the value to add doc names to a new list, count to another new list and word count per doc to another list. As a word is found in each doc, the words in doc count is increased by 1 thus giving the output as key (word, docname) and value as (count of word_indoc, Count of all words in doc, Count of word in Corpus(Alldocs)).

Input : (word ,(docName, count, N, 1))

Function:

```
doc_name= [ ]
Freq = [ ]

Word_count_perdoc = [ ]
Total_docs_count = 0
For each line in Input:
    (word,count) = word.split(",")     #Split line with delimiter(", ") to obtain key,value (word, count)
     words.append(word)               #Append each word into the word list
     freq.append(count)               #Append each count into the freq list
     word_count_perdoc.append(N)
     Total_docs_count += 1  #sum the count to get the number of words in corpus
     Yield output
         #yields output as (word, docname),(count,words_count_perdoc, wordcount_corpus) as key,val pair
```

Output : ((word,docName),(count, N, m))   #m = total word count in corpus(All docs)

MAP REDUCE - 4 (Computer TF-IDF)

Mapper 4:

The Mapper overrides the previous Mapper functions and takes Input from Reducer 3. The Term frequency = (count of word in doc/ Total words in doc) and the Inverse Document Frequency = log(NO.of docs/ count of word in Corpus+1) is calculated whose values are obtained from the Input. Then TFIDF is calculated as a Product of TF and IDF.

Here, The word given is ‘Health’. So, if the word is Health the obtain the tfidf of that word in every doc in the corpus separately.

Input : ((word,docName),(count, N, m))

Function:

```
For each line in the Input:
    TF = val[0] / val[1]              #Compute Term frequency = count of word in doc/ total words in doc
    IDF = log(No.of.docs /( m+1))  #Compute Inverse Document Frequency =
                                          log(No.of docs/ (occurance of word in all docs i.e corpus +1))
    TFIDF = tf * idf                 #Compute tf-idf = Product(Term frequency, Inverse Document Frequency)
    if word == ‘health’:              #if Given word matches to the word (Here, given word is “HEALTH”)
       yield(Output)                   #Yield the ((word required,docname), tf-idf) as key, value pair
```

Output : ((word, docName), tf*idf)

Reducer 4:

Identity Function : Identity function can be defined as that which takes the input and returns the same as output without changing it.

****