Database Management Project: TF-IDF

Clara Dionet, Mirae Kim

❖ Project Overview

Term Frequency-Inverse Document Frequency (TF-IDF) is a statistical measure used to evaluate the importance of a word to a document in a corpus. It is proportional to the number of times a word appears in a document and inversely proportional to the number of times the word appears in the corpus.

The TF-IDF weight is calculated as follows:

$$TF - IDF \quad = \quad TF \, * \, IDF$$

$$= \frac{number\ of\ times\ word\ w\ appears\ in\ a\ document}{total\ number\ of\ words\ in\ the\ document} * log\left(\frac{total\ number\ of\ documents}{number\ of\ documents\ word\ w\ appears\ in}\right)$$

The TF-IDF metric is often used by search engines to determine a document's relevance given a user query. In such applications, computing this metric may involve the manipulation of extensive amount of data which justifies the need for parallelized job processing systems.

In this project, we implement the MapReduce algorithm to calculate the TF-IDF scores given an input set of documents. Different key-value pairs have to be generated in order to ensure the multi-level aggregation of words and documents. We provide both a Python hadoop and Spark solution and compare the performance of each implementation. We also evaluate the processing time of our algorithms according to an increasing input set size: we tested on 5 sets containing 5 equal-sized documents of 5, 10, 20, 40 and 80 MB.

The documents used to calculate the TF-IDF matrices were generated using python. We used several e-books downloaded from the Project Gutenburg website then extracted a set of words from the books, from which we drew at random to create the text in the documents. Because we created the text documents in python, we were more easily able to validate the outputs of the hadoop and the spark implementations as we were able to calculate the TF-IDF metrics directly on the input.

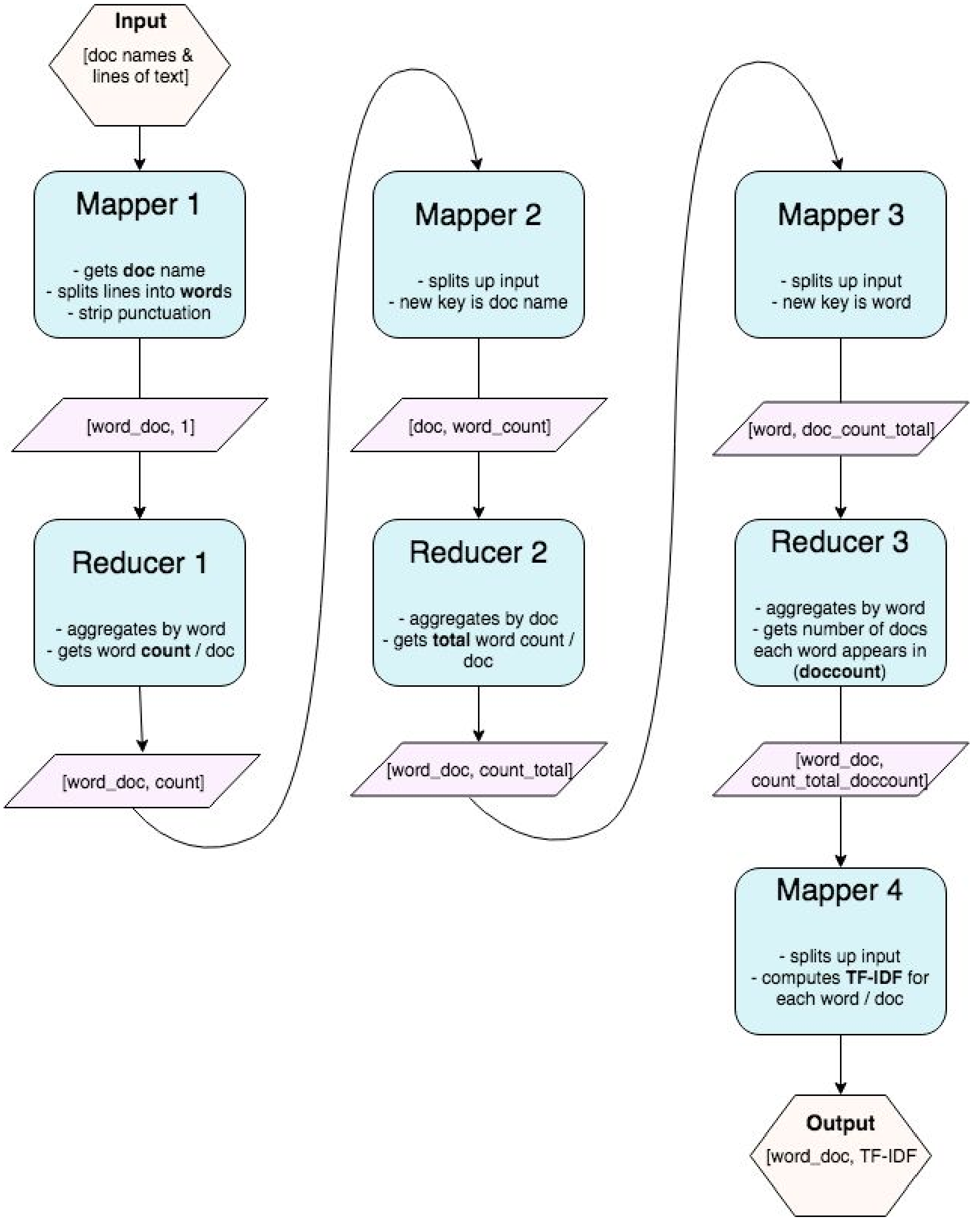
With respect to the data cleansing involved to analyze clean and relevant words, we stripped out both non-alphanumeric characters and stop words from our analysis.

❖ Algorithms

Hadoop

Our hadoop solution consists of 4 Mappers and 3 Reducers run sequentially. Each Mapper takes the output of the previous Reducer as new input, allowing the aggregation to be done on different key-value pairs.

The input consists of various documents with multiple lines of text. The first Map-Reduce computes a word count for each word-document. The second Map-Reduce then counts the total number of words for each document. The third Map-Reduce determines the number of documents each word appears in, and finally the fourth Mapper calculates the TF-IDF of each word per document. A diagram of the algorithm structure is shown below:



Mapper1:

This function splits each line into words and strips all words from any non alphanumeric character. Any words present in a predefined “stopwords” list is also removed. It outputs pairs of concatenated word-document key and value of 1.

```
words = line.split()
for word in words:
    word=word.lower();
    #removing any punctuation
    wordc = ''.join(ch for ch in word if ch.isalnum())
    if (wordc!='' and wordc not in stopwords):
        word_doc=wordc +' '+doc;
        print '%s\t%s' % (word_doc, 1)
```

Reducer1:

This reducer then calculates the word count for each word-document by summing all 1s. The output consists of pairs of (word-document, count).

```
if current_word == word_doc:
    current_count += count
else:
    if current_word:
        # write result to STDOUT
        print '%s\t%s' % (current_word, current_count)
    current_count = count
    current_word = word_doc
```

Mapper2:

This function splits each line into word, document and count respectively. It creates a new key-value pair: (document, word-count).

```
word_doc,count=line.split('\t',1)
word,doc=word_doc.split(' ',1)
word_count=word +' '+count;
print '%s\t%s' % (doc, word_count)
```

Reducer2:

This algorithm then aggregates by document and counts the total number of words each contains. This is done by summing the counts of all words in each document.

```
doc,word_count = line.split('\t', 1)
word,count = word_count.split(' ', 1)
count=int(count)
if prev_doc == doc:
    # total number of words is the sum of each word count
    total=total+count
else:
   if prev_doc != None:
        df[prev_doc]=total
   total=0
   prev_doc = doc
```

Finally, to preserve all previous computations, reducer2 outputs a key-value pair of (word-doc, count-total).

Mapper3

This function splits each line into word, document and count-total. It creates a new key-value pair: (word, doc-count-total-1). The key is the word only, and the value contains a “1” in addition to the document name, word count and total number of words for that document.

```
word_doc,count_total=line.split('\t',1)
   word,doc=word_doc.split(' ',1)
   z=doc+' '+count_total+' '+str(1)
   print '%s\t%s' % (word,z)
```

Reducer3

This function aggregates by word and counts how many documents each word appears in by adding the 1s from the output value of Mapper3.

```
if prev_word == word:
    doccount = doccount+int(c)
```

The following key-value pair is generated: (word-doc, count-total-doccount).

Mapper4

Finally, mapper4 splits each line into word, document and word count and total number of words per document, and doccount (total number of documents each word appears in). With the total number of documents (added manually), all elements are present to compute the TF-IDF for each word.

```
D=5.0
# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    wf,nNm=line.split('\t',1)
    n,N,m=nNm.split(' ',2)
    n=float(n)
    N=float(N)
    m=float(m)
    tfidf= (n/N)*log10(D/m)
    print '%s\t%s' % (wf,tfidf)
```

commands_TFIDF

To run sequentially the algorithms and assess their performance, we constructed a shell file to execute command lines automatically. The time was recorded in between each map-reduce job with the following command:

```
T0=`date +%s`
```

Below shows the code to execute the first two MR functions. Intermediate output directories were created and given as inputs for the following MR job.

```
##Command to execute first map-reduce job
hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar \
-input /user/hadoop/wc/input \
-output /user/hadoop/wc/output_interim \
-file /home/hadoop/mapper1.py \
-mapper /home/hadoop/mapper1.py \
-file /home/hadoop/reducer1.py \
-reducer /home/hadoop/reducer1.py

T1=`date +%s`

##Command to execute second map-reduce job
```

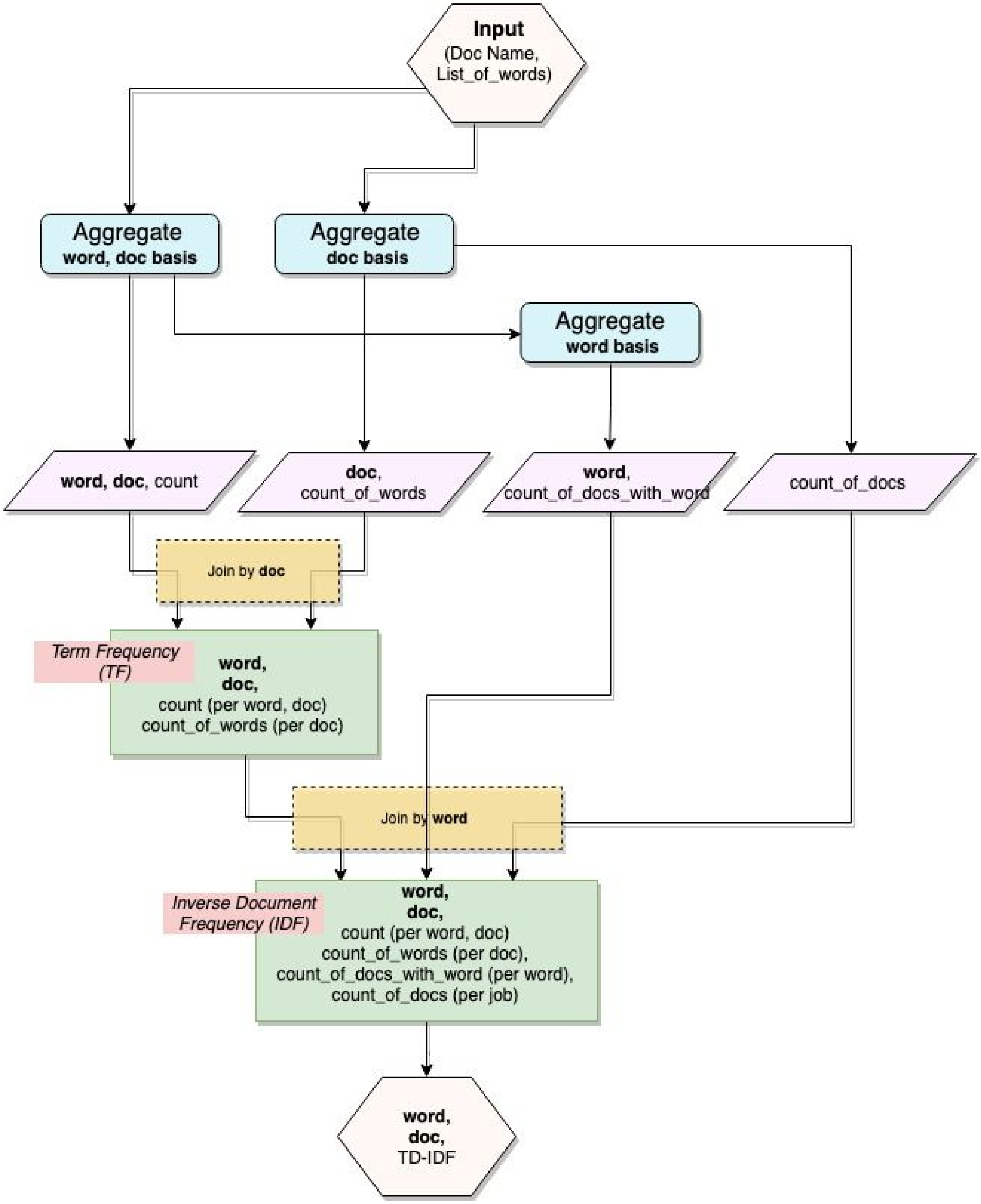
```
hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar \
-input /user/hadoop/wc/output_interim \
-output /user/hadoop/wc/output_interim2 \
-file /home/hadoop/mapper2.py \
-mapper /home/hadoop/mapper2.py \
-file /home/hadoop/reducer2.py \
-reducer /home/hadoop/reducer2.py

T2=`date +%s`
```

Finally, we printed all times at the end of the MR job.

```
echo $T0
echo $T1
echo $T2
echo $T3
echo $T4
```

Spark

The spark algorithm starts by creating a list of tuples, with file name as the first element and a list of words as the second element. The list of words is generated by separating the original text by the space character. The first aggregate that is performed is to count the total number of such tuples, which gives us the number of total documents. Next, we count the number of total words in each document, that is, per document, we count the total number of times a given word appears. We convert this data to a dataframe which we will need later for joining. The final aggregate performed is for each word, we count the number of documents that contains the word. After counting all relevant metrics on a document, word and word+document granularity, the final operations join all data points together to calculate the TF-IDF per word and document. Below is a diagram of this procedure.



The purple parallelograms are the metrics used to calculate the Term Frequency (TF) and the Inverse Document Frequency (IDF). There are three levels of aggregation (labeled “Aggregate”) required to generate the metrics are shown in blue. As shown in the diagram, the word, doc basis aggregation and doc basis aggregation are done directly using the main input data, which are the tuples of file names and list of words in each document. The aggregate on the word basis uses the word, doc basis aggregate the input. On Spark, these parallelograms were saved as data frames (except for the count of docs which was saved as a variable because it is only one number) so that they could be used later downstream for joining purposes.

The green rectangles represent data frames that were created through joining metrics based on the index that is indicated in the yellow boxes. For example, in order to calculate term frequency which requires number of total words in a document and the number of times a certain word appears in a document, we joined the data frame containing count on a word and document basis with the data frame containing total word count on a document basis by the join key document.

Aggregation

In order to load the documents, we used the `wholeTextFiles` method and input the directory path to the documents as a parameter. This instantly creates a list of tuples with file name as the first element and the text in the document as the second element.

```
text_things = spark.sparkContext.wholeTextFiles(input_dir)
```

First, the `text_things` object was stripped from any punctuation and other non-alphanumeric characters, it was then split by the space character to generate a list of words. This object was called `split_word`.

```
split_word = text_things.map(lambda x: (x[0], clean_characters(x[1]).split(' ')))
```

`Split_word` object was used to create two things: the word count by document (`total_words_df`) and the count by word and document (`word_and_file_count_df`). The number of rows in `total_words_df` corresponds with the number of documents in the directory (`total_docs_count`)

```
total_words_df = split_word.map(lambda x: (x[0], len(x[1]))).toDF(['document',
'twc'])
total_docs_count = total_words_df.count()
```

In order to get the count by word and document using the `split_word` object, we needed to change the keys. `Split_word` is keyed by document and has the list of words in the document as the second element. We used list comprehension on this second element, so that every word was a tuple with another inner tuple containing word and document (now the new key) as the first element, and 1 as the second element. This was wrapped inside a `flatMap` to rid the nested list structure and create one large list. Now that we had a list of tuples with word and document as the key, and 1 as the value, `reduceByKey` was used to add the values within each key and create the `word_all_count` RDD. `word_all_count` was also saved as a data frame (`word_and_file_count_df`) so that we could use it for joining later on.

```
word_all_count = split_word.flatMap(lambda x: ([((e,x[0]),1)for e in
x[1]])).reduceByKey(lambda x,y: x+y)
word_and_file_count_df = word_all_count.map(lambda x:(x[0][0], x[0][1],
x[1])).toDF(['word', 'document', 'wc'])
```

The `word_all_count` was also used to calculate per word, the number of documents that word appeared in. Because `word_all_count` was on a word and document basis, we would only have the change the key to contain only word, and count the number of items that existed for each word. We implemented a map to alter the tuple to contain word (new key) as the first element, and 1 as the second element, after which we applied a `reduceByKey` to add all the values within each key.

```
document_word_count_df = word_all_count.map(lambda x:
(x[0][0],1)).reduceByKey(lambda x,y:x+y).toDF(['word', 'doc_count'])
```

Join Operations

There were two join operations used to create the final intermediary data frames used to calculate TF-IDF. The first join operation uses document as the join key using total_words_df (total words per document) and word_and_file_count_df (total count of words per word and document). This data frame (`tf_df`) is then used to calculate the column tf that is the Term Frequency metric, which is stored in the `tf_df_result_df`. Pyspark.sql.functions was used to be able to refer and use columns in dataframes for calculation purposes.

```
import pyspark.sql.functions as F

tf_df = word_and_file_count_df.join(total_words_df,'document','inner')
tf_df_result_df = tf_df.withColumn('tf', F.col("wc")/F.col("twc"))
```

Next, we continued using the `tf_df_result_df` data frame and joined `document_word_count_df` using word as the join key.

```
idf_df = tf_df_result_df.join(document_word_count_df, 'word', 'inner')
idf_df_result_df =
idf_df.withColumn('idf',F.log(total_docs_count/F.col('doc_count')))
```

Finally, the `tf_idf` was calculated by multiplying tf and idf together.

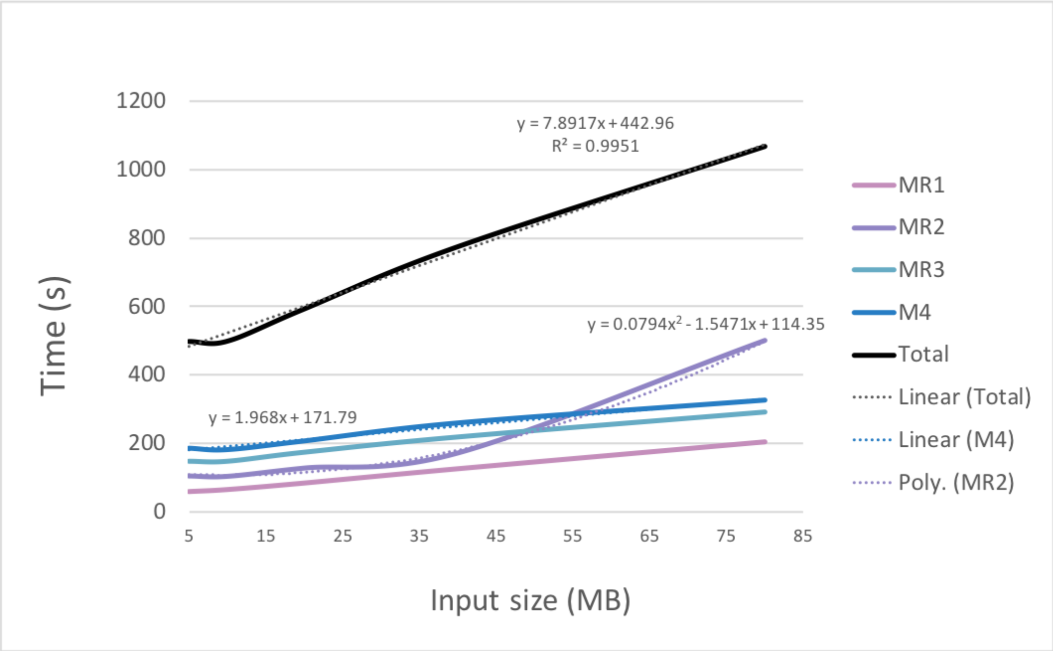
```
tf_idf_result_df = idf_df_result_df.withColumn('tf_idf', F.col('tf')*F.col('idf'))
```

❖ Experimental Analysis

Hadoop:

Below are the timing in seconds of our algorithms according to different input sizes. We tested on 5 sets containing 5 x 5, 10, 20, 40 and 80 MB documents.

Input Size (MB)	*Map Reduce 1 (s)*	*Map Reduce 2 (s)*	*Map Reduce 3 (s)*	*Map 4 (s)*	***Total (s)***
5	*58*	*105*	*149*	*187*	***499***
10	*64*	*104*	*149*	*183*	***500***
20	*83*	*128*	*175*	*208*	***594***
40	*125*	*172*	*219*	*260*	***776***
80	*204*	*500*	*291*	*326*	***1069***

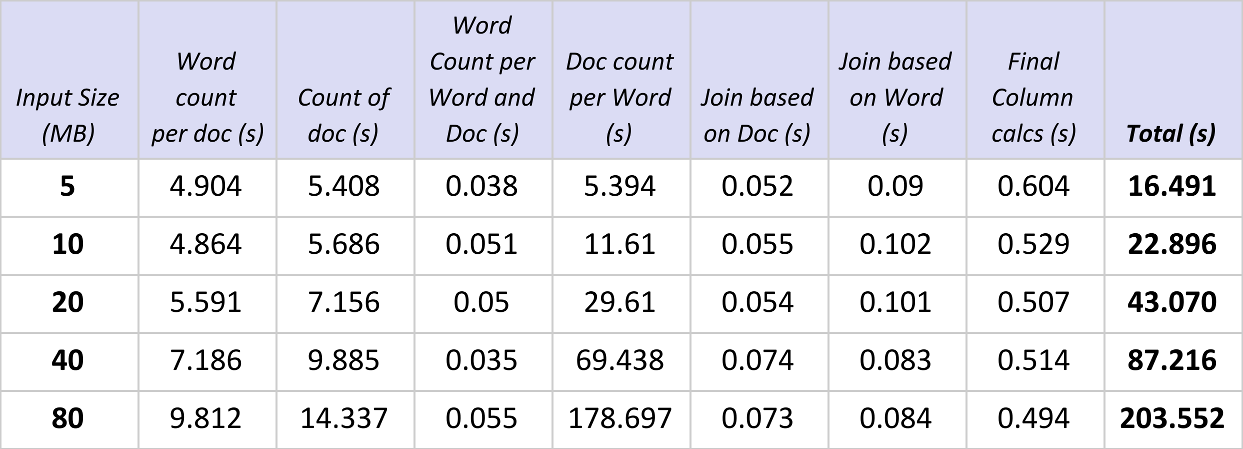


As shown in the graph, the total time for Hadoop Map Reduce increases linearly with the input size. The trendline shows a slope of 7.89, meaning that a difference of 5x10 MB for the document input set leads to 1min20s of additional total processing time. We see however that most of this time is allocated to Map Reduce 2, thus the weak point of our algorithm, responsible for counting the number of words per document. Other Map Reducers have much smaller slopes, such as Mapper 4 which is close to 2: an input increasing by 5x10MB in size only adds 20 seconds for this algorithm.

Spark:

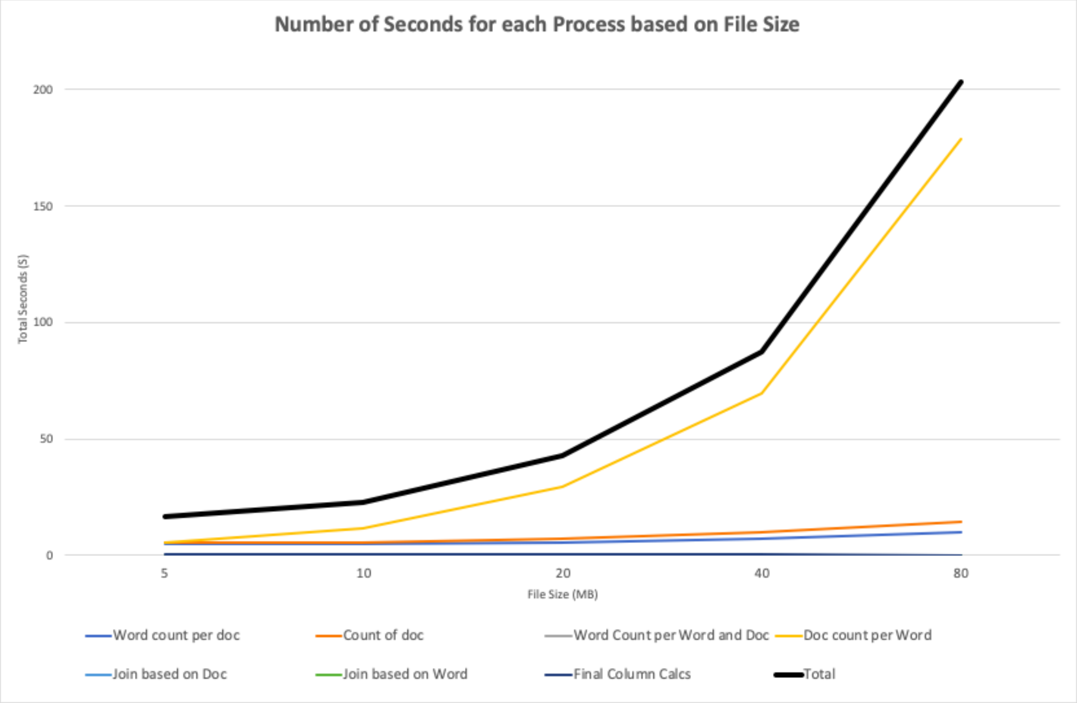
Processing Time Results from initial implementation

Below is a graph of the processing times at each step per document input size. We can see that most processes scale well with respect to size of document, except for the doc count per word, which is the aggregation process that is the biggest bottleneck of the script for all input size documents and contributes the most to total time it takes to process the documents. The second to most time intensive process is count of docs, which is surprisingly heavy even though it only needs to count how many documents there are in the set, which is a very high level aggregation.

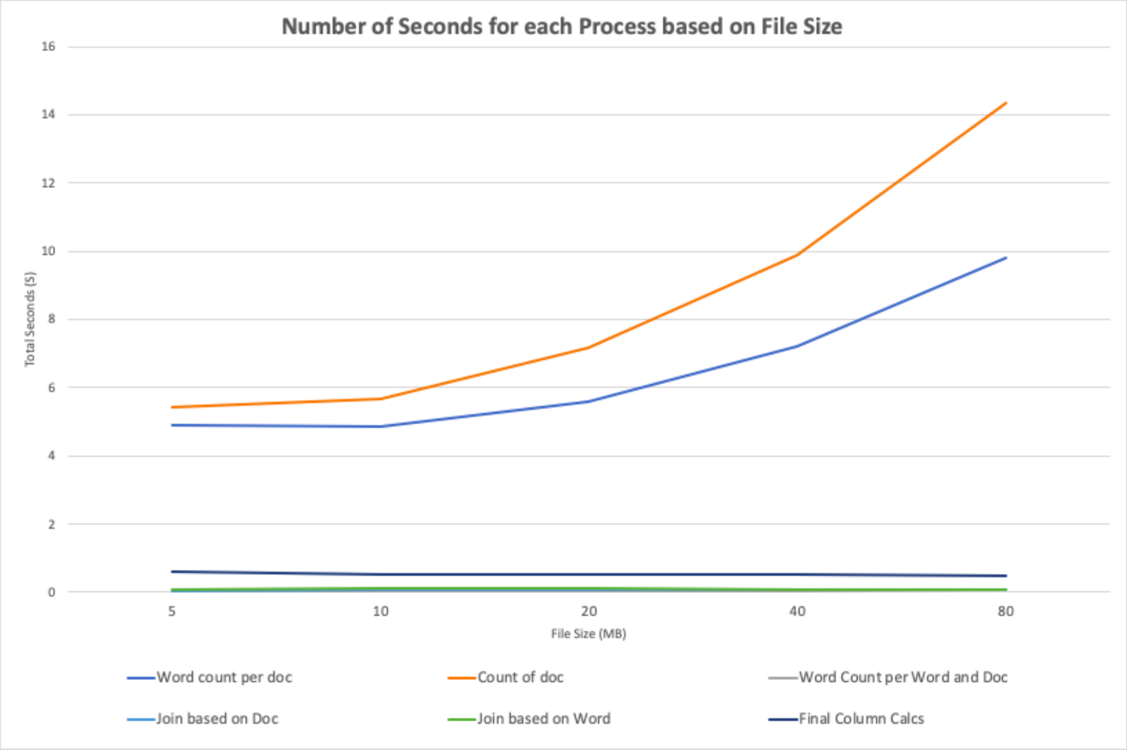


Input Size (MB)	*Word count per doc (s)*	*Count of doc (s)*	*Word Count per Word and Doc (s)*	*Doc count per Word (s)*	*Join based on Doc (s)*	*Join based on Word (s)*	*Final Column calcs (s)*	***Total (s)***
5	4.904	5.408	0.038	5.394	0.052	0.09	0.604	**16.491**
10	4.864	5.686	0.051	11.61	0.055	0.102	0.529	**22.896**
20	5.591	7.156	0.05	29.61	0.054	0.101	0.507	**43.070**
40	7.186	9.885	0.035	69.438	0.074	0.083	0.514	**87.216**
80	9.812	14.337	0.055	178.697	0.073	0.084	0.494	**203.552**

A graphic representation better shows the scalability of the spark implementation with respect to file size. The aforementioned “doc count per word” process increases exponentially with file size, and contributes heavily to the thick black line which shows the total processing time.



Below is another graph with the “doc count per word” and total time removed to more clearly depict the processing times. The scale is completely different from the previous graph, as most of these processes take less than 15 seconds even at the largest file size level. Again, count of doc shows up as the longest process despite it being the highest level aggregation, after which comes word count per doc, the second highest level aggregation. The two join operations and the word count per word and doc do not take up much time at all.



Data Types

Further investigation into the time consumption of the doc count per word showed that what is actually taking up the most time is the doc count per word process was actually the line above it which is transforming the word count per word and doc RDD into a dataframe for joining purposes later down the pipeline. While all operations could have been done with an RDD, we were interested in using the dataframe because it would allow us to structure our data into columns and be able to see the breakdown of the TF and IDF calculations. However, this decision turned out to be more expensive than we initially imagined.

We could consider keeping everything in the RDD state, but we run into the problem of having to do the word-based and document-based join further down the line when combining metrics. RDD joins can be avoided if we transform some of the RDDs into a broadcasted dictionary and do a hash-lookup to achieve a join-like result.

After trying this implementation on the 40MB document set, however, it turned out that even if we can avoid making dataframes, turning RDDs into broadcasted dictionaries can actually take more time than turning RDDs into dataframes. For example, the code to turn word_all_count into a dataframe takes 0.676 seconds.

```
document_word_count_df = word_all_count.map(lambda x:
(x[0][0],1)).reduceByKey(lambda x,y:x+y).toDF(['word', 'doc_count'])
```

But the code to turn it into a dictionary takes 77.887 seconds.

```
document_word_count_dict = dict(word_all_count.map(lambda x:
(x[0][0],1)).reduceByKey(lambda x,y:x+y).collect())
```

Furthermore, we tried using RDD joins instead of dataframe joins which dramatically improved the performance of our process. In the code below, we tested the RDD join on a word basis. RDD joins do require more map operations in general because the join key needs to be the first element of the tuple, so one can end up with a several nested tuples. Below is the code that joins word_and_file_count which as the form (word, (doc, word_in_doc_count)) and document_word_count which has the form (word, count_of_docs_with_word).

```
word_and_file_count = word_all_count.map(lambda x:(x[0][0], (x[0][1], x[1])))
document_word_count = word_all_count.map(lambda x: (x[0][0],1)).reduceByKey(lambda
x,y:x+y)
word_join = word_and_file_count.join(document_word_count)
```

Number of Executors

We tried changing the number of executors from 5 to 20 and ran it on the 80MB document set. Below are the results of the experiment. While overall it was only marginally faster, there are some operations that it performed worse on (Word count per doc and count of doc) and other operations where it performed better (Doc count per word).

	5 Executors	20 Executors	Difference
Word count per doc	9.812	10.515	-0.703
Count of doc	14.337	15.178	-0.841
Word Count per Word and Doc	0.055	0.047	0.008
Doc count per Word	178.697	175.718	2.979
Join based on Doc	0.073	0.069	0.004
Join based on Word	0.084	0.085	-0.001
Final Column Calcs	0.494	0.616	-0.122
Total	**203.552**	**202.229**	**1.324**

Broadcast and Accumulator Variables:

There are a couple of places in the spark implementation where it would be appropriate to use broadcast and accumulator variables. These are variables that can be shared across tasks, which will eliminate the need for us to carry over copies of variables from one task to another.

The accumulator variable is a shared variable that several different workers can contribute to. We can use it to count the number of documents as we have an object which length corresponds to the number of documents. Below is the code that instantiates the accumulator variable (`total_docs_count_accum`). We define a function `file_num` in which we define the mechanism of incrementing the accumulator variable. This is a useful optimization because counting documents is a relatively time consuming process according to our initial analysis.

```
total_docs_count_accum = spark.sparkContext.accumulator(0)

def file_num(x):
        global total_docs_count_accum
        total_docs_count_accum.add(1)
```

`Split_word` is a list of tuples with file name as the first element and the list of words in the second element. We can use the function foreach to call the function `file_num` for each element in split_word, which should correspond to the total number of documents.

```
split_word = text_things.map(lambda x: (x[0], clean_characters(x[1]).split(' ')))
split_word.foreach(file_num) #count the num of docs
```

After this operation, the `total_docs_count_accum` will now have the correct number of docs.

The broadcast variable is a read-only variable that is sent to each node's cached memory. It can be used to replace some of the join operations that we previously used. We transformed two objects into broadcasted dictionaries instead of dataframes. These were the `total_words_dict`, which contains the total word count per document and the `document_word_count_dict`, which contains per word, the number of docs that word appears in.

```
total_words_dict = dict(split_word.map(lambda x: (x[0], len(x[1]))).collect())
#create a broadcast variable
bc = spark.sparkContext.broadcast(total_words_dict)

document_word_count_dict = dict(word_all_count.map(lambda x:
(x[0][0],1)).reduceByKey(lambda x,y:x+y).collect())
document_word_count_bc = spark.sparkContext.broadcast(document_word_count_dict)
```

We were able to use these broadcasted dictionary variables instead of performing two dataframe joins by simply doing a hash lookup with document and word.

```
word_and_file_count_df = word_all_count.map(lambda x:(x[0][0], x[0][1], x[1],
bc.value[x[0][1]], document_word_count_bc.value[x[0][0]])).toDF(['word',
'document', 'wc', 'twc', 'doc_count'])
```

Below are the results of the improved implementation of spark. Unexpectedly, for the 80 MB, the use of broadcast and accumulator variables result in slower processing time.

Input Size (MB)	*With Join Statements (Original)*	*Spark with Broadcast and Accumulator*
5	16.491	11.328
10	22.896	18.734
20	43.07	36.371
40	87.216	81.279
80	203.552	213.036

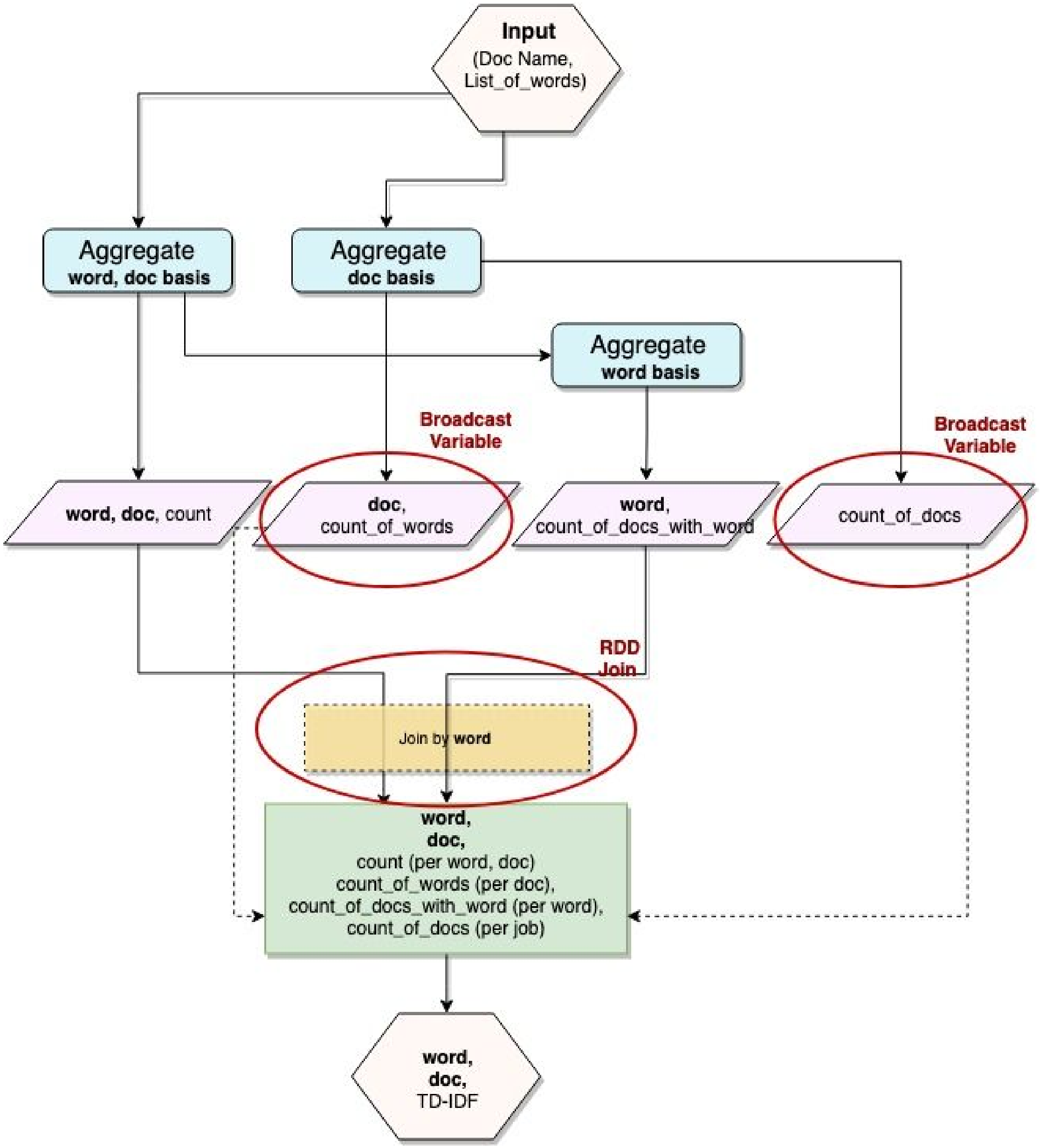
Conclusion

In conclusion, we discovered many points in which we can optimise our code and created a new implementation where we used a mix of the features we found helped improve our performance. We've included this implementation in the appendix (spark_best_implementation.py).

The new implementation performs the word basis join with an RDD join (instead of the original data frame join) and the doc basis join with a hash lookup using a broadcasted dictionary. The count_of_docs is also saved as a broadcasted variable. Below is a diagram of the improved implementation, with the changes highlighted in red: the two new broadcasted variables and the RDD join.

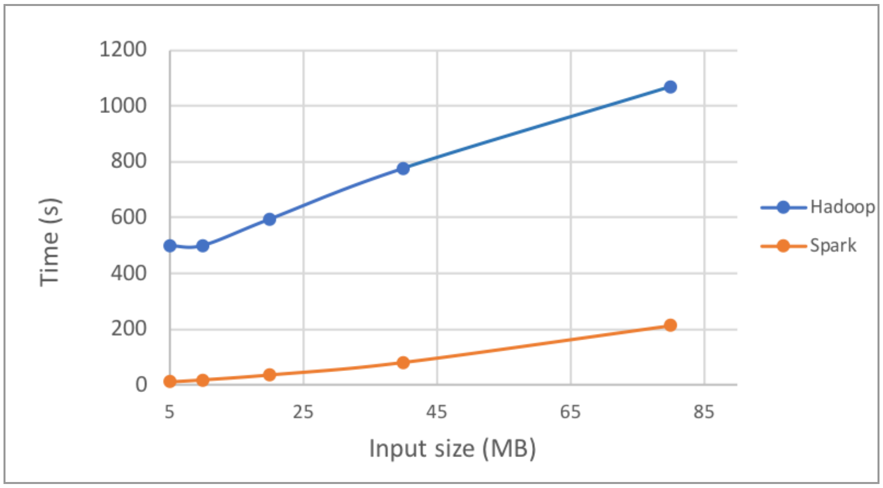
Below are the results of the new implementation versus the original implementation. While we see an improvement of a couple of seconds for each file size category, the results are not a drastic improvement over the original implementation, especially for the 80 MB size.

Input Size (MB)	**Original Implementation (s)**	**New Implementation (s)**
5	16.491	11.101
10	22.896	17.496
20	43.070	36.658
40	87.216	80.742
80	203.552	206.507



❖ Conclusion: Hadoop and Spark Comparison

The graph below shows the total time taken for both implementations in terms of input size (Hadoop and main Spark code).



It is clear that Spark outperforms Hadoop by far in processing speed. As it stores intermediate data in-memory, Spark requires only two accesses to disk (input/output) in comparison to six with Hadoop (I/O of Map, S&S and Reduce) in a typical Map Reduce job. In our case, the Hadoop solution we adopted consisting of 3xMR and 1 additional Mapper requires 20 disk accesses which explains the elongated processing time. Reducing the number of read/write cycle to disk allows Spark to be up to 100x faster in memory and 10x faster on disk than Hadoop. However, its performance may degrade if data does not fit in the memory. These two implementations have different applications as Hadoop is designed to process huge volumes of data whereas Spark is better suited for real time data.

❖ Appendix

List of files attached:
Mapper1.py, Reducer1.py: Step 1 of Hadoop implementation: computes word count/ word-doc
Mapper2.py, Reducer2.py: Step 2 of Hadoop implementation: computes total word count / doc
Mapper3.py, Reducer3.py: Step 3 of Hadoop implementation: computes doc count / word
Mapper4.py: Step 4 of Hadoop implementation: computes TF-IDF / word-doc
Commands_TFIDF.sh: Commands to execute Hadoop code
Doc_generator.py: Python script that generates input document text
Spark.py: Main Spark implementation code
Spark_bc_accum.py: Spark implementation with Broadcast and accumulator variables
Spark_best_implementation.py: Optimised implementation of Spark code