Implementation of TF-IDF calculation using MapReduce

Term Frequency (TF):

Term Frequency also called as TF. It calculates the number of times a word (term) arises in a document. The terms and their frequency on each of the document is measured by the term frequency.

Inverse Document Frequency (IDF):

The search concept is to find out applicable documents similar the query. To find certain words that arise too repeatedly have slight power in defining the significance.

The logarithm method is used for calculating the IDF.

Question 1: (Building Index) Compute TFIDF scores for all words in all documents and build an inverted index?

Answer 1:

Mapper to compute TF:

```
#!/usr/bin/env python
import sys
import os
def termfreq():
  for line in sys.stdin:
    words = line.strip().split() # splitting into line and removing whitespaces
    for word in words:
      print '%s\t%s\t1' % (word,os.getenv('mapreduce_map_input_file','noname')) #
printing frequency with filename
if __name__ == '__main__':
  termfreq()
```

Reducer to Compute TF:

```
#!/usr/bin/env python
import sys
def termreducer():
  currentterm = None
  currentcount = None
  for line in sys.stdin:  # reading file from standard input
    word,filename,count = line.strip().split('\t')  # splitting and removing white
spaces
    term = '%s\t%s' % (word,filename) #printing filename and words
    if currentterm == None:
      currentterm = term
      currentcount = eval(count) #counting word frequency
    elif currentterm == term:
      currentcount += eval(count)
    else:
      print '%s\t%s' % (currentterm,currentcount) #printing current word with
frequency
      currentterm = term
      currentcount = eval(count)
  print '%s\t%s' % (currentterm,curcount)
if __name__=='__main__':
  termreducer()
```

Document Frequency Mapper and Reducer: The output from the term frequency map reduce was used as input for calculating the document frequency.

```
#!/usr/bin/env python
import sys
import os
def docfreq():
  for line in sys.stdin:
    print "%s\t1" % line.strip() # reading the input file and  removing whitespaces
if __name__ == '__main__':
  docfreq()

#!/usr/bin/env python
import sys
def docfreqred():
  currentword = None
  currentcount = None
  space = []  # list/dic for count storage
  for line in sys.stdin:
      word,filename,wordcount,count = line.strip(',').split(',') # splitting into
lines
     term = "%s\t%s\t%s" %(word,filename,wordcount) # printing the words along with TF
and filename
    if word == None:
      currentword = word
      currentcount = eval(count)  #counting across documnets
      space.append(term)
    elif currentword == word:
      currentcount += eval(count)  #incrementing the count
      space.append(term)
    else:
      for item in space:
        print "%s\t%d" % (item,currentcount)
      currentword = word
      currentcount = eval(count)
      space = [term]
  for item in space:
    print "%s\t%d" % (item,currentcount)    #printing the results
if __name__=='__main__':
  docfreqred()
```

TF_IDF calculation: For calculating the TF-ID, the output of the previous map reduce file inclusing term frequency and document frequency. Only the mapper will be needed to perform the operation.

```
#!/usr/bin/env python

import sys
import os
from math import log10,sqrt

num_docs = 265.0 #total documnets
# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove whitespace
    line = line.strip()
    # split the line into words
    word,ndf=line.split('\t',1)
    termfreq,df = ndf.split(' ',2)
    n=float(n) # freq of term in doc
    #N=float(N) #sum of doc with terms
    df = float(df) #doc freq
```

```
    tfidf= (n)*log10(num_docs/df)
    print '%s\t%s' % (word,tfidf)
```

TF Score Print Screen:

```
spells  noname  2
spending        noname  1
spend   noname  2
spent." noname  1
spent   noname  6
spice   noname  2
Spicer  noname  1
spices  noname  1
spicy   noname  1
spilled noname  3
spilling        noname  1
spine   noname  1
spin    noname  1
spiralling      noname  1
spirit,"        noname  1
spirit, noname  1
spirit! noname  1
spirit  noname  9
splashing       noname  1
split-second    noname  1
SPL     noname  1
spoil   noname  1
spoken  noname  8
spoke   noname  8
```

Question 2: Search Query:

I did this question on spark because spark is more efficient in real time results of massive data then the MapReduce.

Pyspark code:

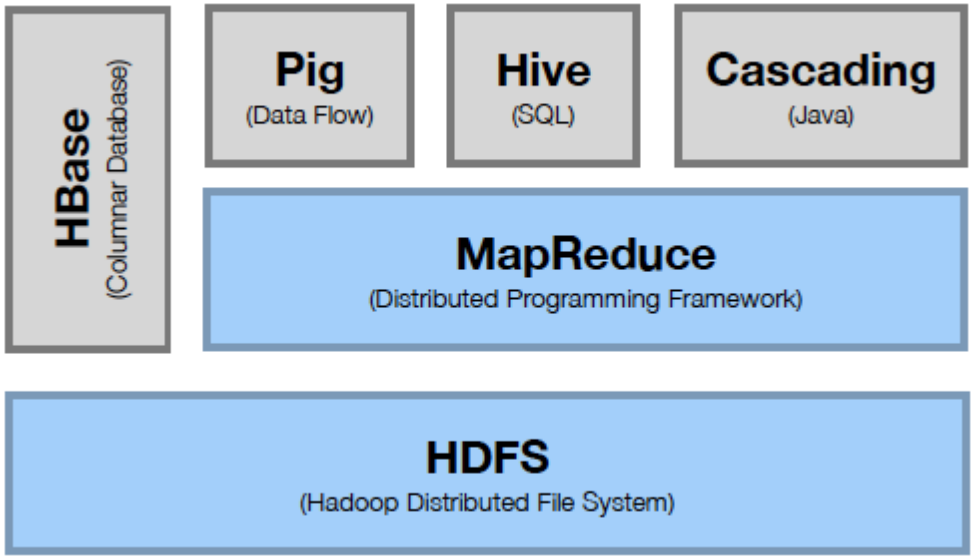
```
from pyspark.sql import SQLContext, Row
from pyspark.ml.feature import Tokenizer, RegexTokenizer
from pyspark import SparkContext, SparkConf
from numpy import np

def main(sc):
    tfidfFile = sc.textFile("/user/root/tfidf_output").map(lambda x: (x['term'],
(x['doc'], x['score'])))  # the file with tfidf scores

#search function to find Top N:
def search(query, topN):
    searchquery = sc.parallelize(Tokenizer(query)).map(lambda x: (x,
1)).collectAsMap()
    broadTokens = sc.broadcast(searchquery) # filter out the interesting documnets
```

```
    jointfidf = tfidfrdd.map(lambda a, b: (a, broadTokens.value.get(a, '-'),
b)).filter(lambda a, b, c: b != '-')

   # aggregateByKey to compute the similarity score:
    searchCount = jointfidf.map(lambda a: a[2]).aggregateByKey((0, 0),
                                                               (lambda V, value: (V[0] +
value, V[1] + 1)),
                                                               (lambda V1, V2: (V1[0] +
V2[0], V1[1] + V2[1])))

    scores = searchCount.map(lambda a, b: (a[0] * b[1] / len(searchquery),
a)).top(topN)

    print(scores)

if __name__ == "__main__":
    conf = SparkConf().setAppName("MyApp")
    sc = SparkContext(conf = conf)
    main(sc)
    sc.stop()


#spark-submit --master yarn-client --executor-memory 512m --num-executors 3 --
executor-cores 1 --driver-memory 512m SearchSpark.py
```

Summary:

The technology I pick is MapReduce job run on the Hadoop clusters. Hadoop map reduce is a software framework for writing the application which processes vast amount of data in parallel on large clusters. Map reduce job usually splits input data into independent chunks which are processes by map task independently.

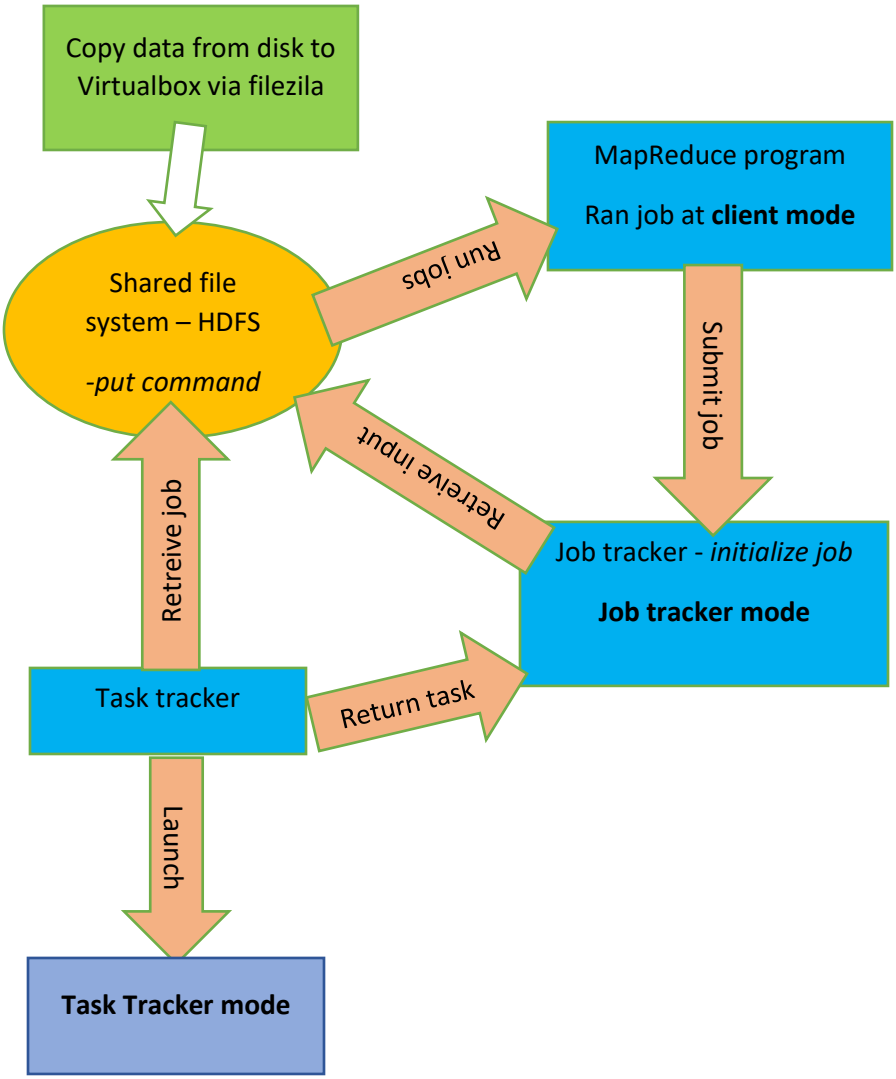
MapReduce is a computational model that decomposes large chunks of data jobs into individual clusters that can be executed parallel across a cluster.



MapReduce is reliable and fault tolerance, no synchronization and no network code is needed.

MapReduce is faster is bactch processing of big data. It stores Data on the disk and Mapreduce is basically written in Java while spark is good and relaible for real time data processing and it stores data into the memory.

Flow Chart:



References:

1. https://spark.apache.org/examples.html , retrieve at Dec 7-2017
2. https://www.edureka.co/blog/apache-spark-vs-hadoop-mapreduce , Retreived at Dec 7- 2017
3. https://spark.apache.org/docs/2.1.0/ml-features.html#tf-idf, Retreived at Dec 7- 2017