TF-IDF Based Querying System Using Spark

Amir Ghaderi – 500794236

December 8th, 2016

High Level Overview

The following document is used to explain the "*TF-IDF Based Querying System Using Spark".* The system is divided into two spark submit files, the first creates a TF-IDF index and the second runs a query on that index. This document will break down and explain each of the spark submit files.

Part 1 – Creating the TF-IDF Index (TF-IDF.py)

a) The following block of code iterates through each text file located in the HDFS directory "/user/amir/football/*" and performs a basic word count. Each of the resulting RDDs are then merged together to create a master RDD. This master RDD contains all unique terms from each individual text file. Please note that the reason for the multiple if statements is merely for the way the text files are named.

```
for i in range(1,266):
    if i==1:
        #Convertes i into a string so it can be concatenated into a file name
        i = str(i)
        #String concatenation
        file_location = "/user/amir/football/00" + i + ".txt"
        #Loads RDD
        text = sc.textFile(file_location)
        #Splits on spaces
        words = text.flatMap(lambda line: line.split())
        #Creates word counter
        wordWithCount = words.map(lambda word: (word, 1))
        #Basic word count
        final_rdd = wordWithCount.reduceByKey(lambda v1,v2: v1+v2)
    elif i > 1 and i < 10:
        i = str(i)
        file_location = "/user/amir/football/00" + i + ".txt"
        text = sc.textFile(file_location)
        words = text.flatMap(lambda line: line.split())
        wordWithCount = words.map(lambda word: (word, 1))
        count = wordWithCount.reduceByKey(lambda v1,v2: v1+v2)
        #Union of the first RDD with all other RDDs, note RDD union doesn't remove duplicates
        final_rdd = sc.union([final_rdd,count])
    elif i  >=10 and i <100:
        i = str(i)
        file_location = "/user/amir/football/0" + i + ".txt"
        text = sc.textFile(file_location)
        words = text.flatMap(lambda line: line.split())
        wordWithCount = words.map(lambda word: (word, 1))
        count = wordWithCount.reduceByKey(lambda v1,v2: v1+v2)
        final_rdd = sc.union([final_rdd,count])
    elif i >=100:
        i = str(i)
        file_location = "/user/amir/football/" + i + ".txt"
        text = sc.textFile(file_location)
        words = text.flatMap(lambda line: line.split())
        wordWithCount = words.map(lambda word: (word, 1))
        count = wordWithCount.reduceByKey(lambda v1,v2: v1+v2)
        final_rdd = sc.union([final_rdd,count])
```

b) The following block of code creates an IDF score for each word. The first line of code takes the "final_rdd" previously computed and resets all of the counts to 1. The resulting RDD is then applied to a reduceByKey method, which outputs the number of documents each word appears in. Finally, the IDF formula is used to compute the final IDF values.

```
#Reset the count of each word in the Master RDD
wordWithCount = final_rdd.map(lambda x:(x[0],1))
#Word Count on the reset Master RDD
count = wordWithCount.reduceByKey(lambda v1,v2: v1+v2)
#Apply the IDF formula to all values of the Master RDD
final_rdd2 = count.map(lambda x:(x[0],round(math.log(float(265)/float(x[1])),5)))
```

c) The following block of code again iterates through each of the text files located in the directory "/user/amir/football*" and performs a basic word count. However, this time the resulting RDD is merged with the IDF scores RDD and then a reduceByKey method is used to compute the final TF-IDF scores. The outputs are then saved into a new text file directory.

```
for i in range(1,266):
    if i < 10:
        i = str(i)
        file_location = "/user/amir/football/00" + i + ".txt"
        text = sc.textFile(file_location)
        words = text.flatMap(lambda line: line.split())
        wordWithCount = words.map(lambda word: (word, 1))
        count = wordWithCount.reduceByKey(lambda v1,v2: v1+v2)
        #Union with the file containing the IDF scores
        merge = sc.union([count,final_rdd2])
        #Reduce by Key using multiplication
        count2 = merge.reduceByKey(lambda v1,v2: v1*v2)
        #Saving the output as textfile
        count2.saveAsTextFile("/user/amir/final/output/part" + i)
    elif i  >=10 and i <100:
        i = str(i)
        file_location = "/user/amir/football/0" + i + ".txt"
        text = sc.textFile(file_location)
        words = text.flatMap(lambda line: line.split())
        wordWithCount = words.map(lambda word: (word, 1))
        count = wordWithCount.reduceByKey(lambda v1,v2: v1+v2)
        merge = sc.union([count,final_rdd2])
        count2 = merge.reduceByKey(lambda v1,v2: v1*v2)
        count2.saveAsTextFile("/user/amir/final/output/part" + i)
    elif i >=100:
        i = str(i)
        file_location = "/user/amir/football/" + i + ".txt"
        text = sc.textFile(file_location)
        words = text.flatMap(lambda line: line.split())
        wordWithCount = words.map(lambda word: (word, 1))
        count = wordWithCount.reduceByKey(lambda v1,v2: v1+v2)
        merge = sc.union([count,final_rdd2])
        count2 = merge.reduceByKey(lambda v1,v2: v1*v2)
        count2.saveAsTextFile("/user/amir/final/output/part" + i)
```

Part 2 – Querying (Query.py)

a) The following block of code takes a string (query) and an integer (number of documents requested) as input and returns the top n documents with the highest TF-IDF scores. The code iterates through the TF-IDF index directory and computes a score for each document based on the given query. The scores for each document are then stored in a dictionary.

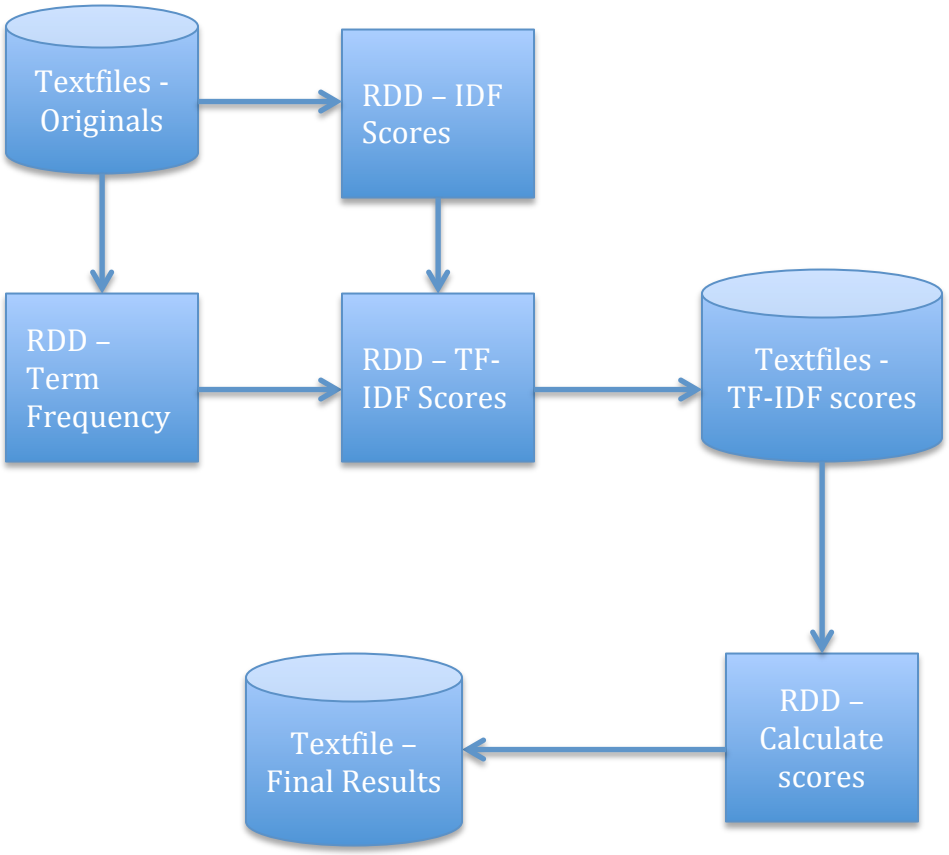
```
def main(sc):
    #Number of documents you wish to retrieve
    n =10
    #The query you wish to search
    query = "What an amazing goal"
    #Splitting the query into a list of strings
    query = query.split()
    #Storing the length of the query
    length = len(query)
    #initializing an empty dictionary
    score_board = {}
    #Iterating through all files and computing a total score per file
    for i in range(1,266):
        #initializing a score for the current file
        score = []
        #initializing the total number of matches per file
        matches = 0
        #initializing a score total
        total = 0
        #Converting i to a string
        i = str(i)
        #grab the textfile
        file_location = "/user/amir/final/output/part" + i
        #convert the textfile into a RDD
        text = sc.textFile(file_location)
        #iterate through each word in the query you are using
        for j in query:
            #Check if the file contains the word
            count = text.filter(lambda x: j in x)
            if count.count() >= 1:
                #Increment matches
                matches = matches + 1
                #Retrieve the score
                word_score = count.map(lambda x: x[-19:-10])
                #Turn the score into a float
                float_score = word_score.map(lambda x: float(x))
                #append the score to a list
                score.append(float_score.take(1))

        #iterate through all scores in the list and sum them
        for p in score:
            total = total + p[0]
        #Multiple the summed score by the fraction of matches/length
        total = total * float(matches)/float(length)
```

```
#Append the document name (key) and final score (value) to dictionary
score_board["Document Number = " + i] = total
```

At this point the scores for each of the documents are stored in the dictionary "*score_board*". The dictionary is then sorted by value in descending order and transformed into a list. The list is then used to retrieve the top n documents with their respective scores. The final results are then exported into HDFS as a textfile.

```
#Sort dictionary
final_score_board = sorted(score_board.items(), key=operator.itemgetter(1))
#Reverse the sort
final_score_board.reverse()
#Grab the first n document
results = final_score_board[0:n]
#Turn the object into a RDD
result = sc.parallelize(results)
#Save the output as a textfile
result.saveAsTextFile("/user/amir/final/answer")
```

Architecture Diagram



Example Executions

Query1 : "What an amazing goal"

N = 1
('Document Number = 122', 20.520879900000001)
N = 3
('Document Number = 122', 20.520879900000001)
('Document Number = 205', 19.827730000000003)
('Document Number = 189', 19.827730000000003)
N = 5
('Document Number = 122', 20.520879900000001)
('Document Number = 205', 19.827730000000003)
('Document Number = 189', 19.827730000000003)
('Document Number = 174', 19.323179799999998)
('Document Number = 207', 19.323179799999998)

Query2 : "That was a close game"

N = 1
('Document Number = 113', 27.985329499999999)
N = 3
('Document Number = 113', 27.985329499999999)
('Document Number = 264', 25.7014295)
('Document Number = 99', 24.602819500000003)
N = 5
('Document Number = 113', 27.985329499999999)
('Document Number = 264', 25.7014295)
('Document Number = 99', 24.602819500000003)
('Document Number = 98', 24.602819500000003)
('Document Number = 95', 24.602819500000003)

Conclusion

Spark was chosen as the tool for this project mainly because of it's processing power. Spark allows for processing data in memory and thus able to out perform other data processing tools i.e. Map Reduce. Unfortunately, the current implementation of the TF-IDF querying system is not fully optimized. The system is currently unable to execute queries in real time. Future implementations/modifications to this system should attempt to hold the TF-IDF index in memory and therefore avoiding reading from disk. This modification would hopefully allow the system to operate in real time.