BBC Sport Search

# Summary – Try to add one more point

* The search system consists of two scripts: “buildIndex.py” and “search.py”.
  + The first code calculates the TFIDF value of each keyword and store results as a parquet file. We can consider this as the database for our search.
  + The second code let’s the user to search the database created by the first script. User can specify the query and the number of results they want to retrieve directly from the terminal.
* The system employs Spark and Spark SQL in the background and does not utilize a database to store the indexed data.
* The system is ideal for a small database where we can fit all of its content into memory.

# Code Summary

This document goes over the key codes in the scripts, for details about the code please check the code section or script accompanying this document.

### How to Submit the Codes?

spark-submit --master yarn --executor-memory 512m --num-executors 3 --executor-cores 1 --driver-memory 512m buildIndex.py

spark-submit search.py -N 5 -Q Open Air Play

[The second algorithm is **not** case sensitive]

### buildIndex.py

Line 17 – 19: Formats the path of the files so that irrelevant parts of the path is removed and it becomes easier to read. The extracted words are also formatted so that all of them are lower case and punctuations are removed.

Line 22 – 23: Load the data into a Spark DataFrame and then flatten the dataset by extracting the words from list.

Line 25 – 26: Calculate the TF values for each word.

Line 28 – 29: Calculate the IDF values for each word.

Line 31 – 32: Joins the DataFrames where we calculated TF and IDF values and multiplies them.

### search.py

Line 35 – 41: Create a parser so that user can provide values from the terminal. The user can provide 3 parameters:

* -Q: the query the user wants to make.
* -N: How many values will be returned by the query. Default is 1.
* -F: The name of the file users wants to read. Default is the output of buildIndex.py

Line 18: Filters the loaded database based on the provided query. Preprocesses the query so that all the letters are lower case.

Line 20: Groups filtered keywords based on the path of the document and totals their TFIDF score.

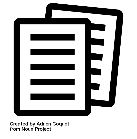
Line 22 – 23: Counts how many of the keywords are captured in the documents and the calculates the fraction of the keywords that are captured.

Line 25 – 26: Accumulates the scores by multiplying the total TFIDF score with the fractions. Then the paths are ordered based on their score and top N values are returned.

Line 28 – 32: The user can alter the code to write the results to a csv file. Currently, by default the user only sees the result in the terminal.

# Diagram

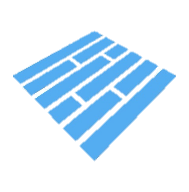
buildIndex.py



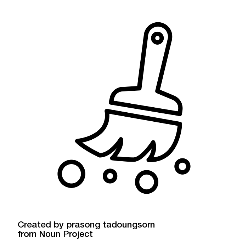
BBC Dataset

HDFS

SQLContext



Spark



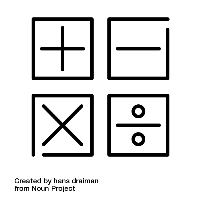
Preprocess data

TFIDF calc.

Parquet

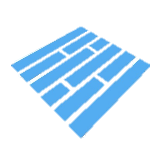
Write

Read

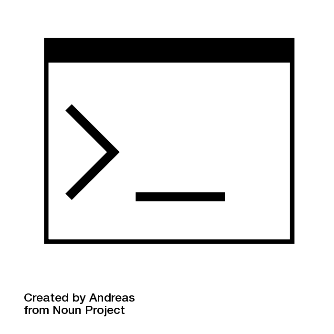


search.py

HDFS



Parquet



Terminal

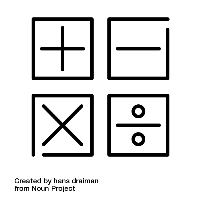
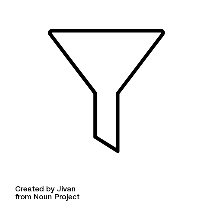
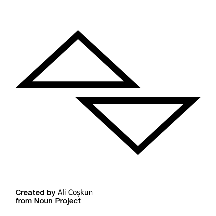
Return

Read

Calc. Score

Filter

Sort



Spark

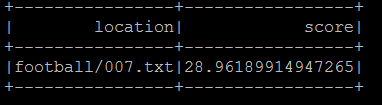
# Why Spark & Spark SQL?

The reason I have picked Spark over other tools is its ease of implementation. The platform handles the parallelization and MapReduce in the background while the user can enter basic SQL commands. Additionally, Professor Miranskyy pointed out that for basic queries, like the ones required for TFIDF, the overhead associated with using the SQLContext is minimal. As a result, I did not use Pig or a MapReduce implementation from HDFS. I could have used Hive since it provides a SQL interface to HDFS but compared to Spark it is less efficient. Unless we use Tez, Hive would read and write files every time we set a MapReduce job. Hence, it is not suitable for our task.

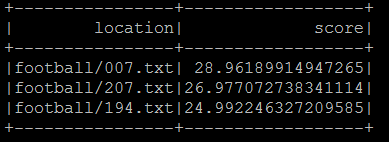
One of the drawbacks of my implementation is since Spark does in-memory computation, it needs to load the data every time the user does a search. For this task, loading the data did not consume a lot of time but for a large dataset (such as collecting keywords from the internet just like Google’s) this step would be computationally expensive. A work around can be starting a server that continuously runs a Spark context and never shuts down. I tried to implement HBase in the Hortonworks Sandbox, but I failed to even start the server. I have not used MongoDB because since it is not best when it comes to establishing relations documents hence it would most likely be a more computationally expensive process to replicate the result in this medium.

# Results

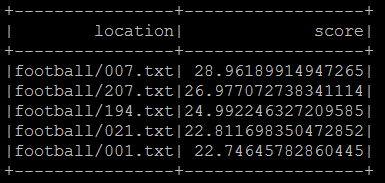
Query: *Manchester United*

N = 1

N = 3

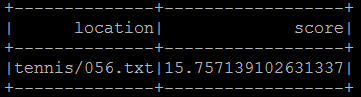


N = 5

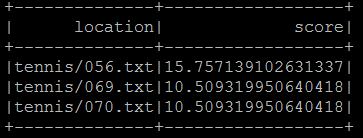


Query: *Open Air Play*

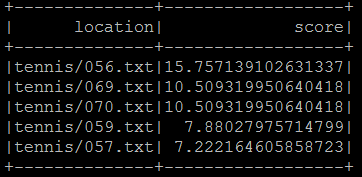
N = 1



N = 3



N = 5



# Code

## buildIndex.py

1. # spark-submit --master yarn --executor-memory 512m --num-executors 3 --executor-cores 1 --driver-memory 512m buildIndex.py
3. **import** re
4. **from** pyspark **import** SparkConf, SparkContext
5. **from** pyspark.sql **import** SQLContext
6. **from** pyspark.sql.functions **import** explode, sum, column, log
7. **from** pyspark.sql.types **import** IntegerType
9. **def** main():
10. conf = SparkConf().setAppName("Index Builder") # configure Spark
11. sc = SparkContext(conf = conf)   # start Spark Context with the specific configuration
12. sql = SQLContext(sc) # start Spark SQL
14. text = sc.wholeTextFiles("/user/root/bbcsport/\*") # fuzy read: Reads all files under bbcsport
15. fileCount = text.count()
16. # reformat data to make it cleaner and break text into words
17. cleaned = text.map(**lambda** file: ("%s/%s" % (file[0].split("/")[len(file[0].split("/"))-2],\
18. file[0].split("/")[len(file[0].split("/"))-1]), file[1].lower().split()))\
19. .map(**lambda** file: (file[0], [re.sub(r'[^\w\s]', '', word) **for** word **in** file[1]]))
20. # regex cleaning from: https://stackoverflow.com/questions/265960/best-way-to-strip-punctuation-from-a-string-in-python
22. cleanedDF = cleaned.toDF(["location", "words"]) # create dataframe
23. cleanedDF = cleanedDF.select(cleanedDF.location, explode(cleanedDF.words).alias("word")) # Flatten the list of words
25. tfMap = cleanedDF.groupby(cleanedDF.location, cleanedDF.word).count() #Count occurences of a word in a document
26. tfReduce = tfMap.groupby(tfMap.location, tfMap.word).agg(sum("count").alias("tf")) # Calculate TF
28. idfMap = cleanedDF.distinct().groupby(cleanedDF.word).count() # count whether a word occured in a document
29. idfReduce = idfMap.select(idfMap.word, log(fileCount/(column("count"))).alias("idf")) # Calculate IDF
31. joinTfIdf = tfReduce.join(idfReduce, tfReduce.word == idfReduce.word, "inner") # Join TF & IDF tables
32. tfIdf = joinTfIdf.select(column("location"), tfReduce["word"], (column("tf") \* column("idf")).alias("tfIdf")) # Calc. TFIDF
34. tfIdf.write.parquet('bbc.parquet') # write file in an efficient file format
36. **if** \_\_name\_\_ == "\_\_main\_\_":
37. main()

## search.py

1. **import** argparse
2. **from** pyspark **import** SparkConf, SparkContext
3. **from** pyspark.sql **import** SQLContext
4. **from** pyspark.sql.functions **import** sum, column, desc
5. **from** pyspark.sql.types **import** IntegerType
7. **def** main(Number, Query, File, write=False):
8. **print**("\nSetting up the enviroment\n")
9. conf = SparkConf().setAppName("Searcher") # configure Spark
10. sc = SparkContext(conf = conf)    # start Spark Context with the specific configuration
11. sql = SQLContext(sc) # start Spark SQL
13. **print**("\nLoading the data\n")
14. data = sql.read.load(File)
16. **print**("\nQuerying the keywords in the database\n")
17. totKeyword = len(Query)
18. filtered = data.filter(column('word').isin([word.lower() **for** word **in** Query])) # Query the database based on request
20. sumed = filtered.groupby(filtered.location).agg(sum('tfIdf').alias("tot"))  # Sum the TFIDF scores
22. counted = filtered.groupby(filtered.location).count()
23. counted = counted.select(counted.location.alias("loc"), (column("count")/totKeyword).alias("freq")) # determine the weight for each word
25. result = sumed.join(counted, on = sumed.location==counted.loc, how = "inner") # join the tables
26. result = result.select(result.location, (column("tot") \* column("freq")).alias("score")).orderBy(desc("score")).limit(Number) # Calculate score and return top N values
28. **if** write:
29. **print**("\nWriting the data\n")
30. result.write.format('com.databricks.spark.csv').save('query\_'+''.join(Query), header='true')
31. **else**:
32. result.show()
34. **if** \_\_name\_\_ == '\_\_main\_\_':
35. parser = argparse.ArgumentParser(prog="searcher", add\_help=False)
36. parser.add\_argument("-N", default = 1, type = int,
37. help= 'Picks the top N documents that match the query')
38. parser.add\_argument("-Q", type= str, required=True, nargs='\*',
39. help= 'User provided query')
40. parser.add\_argument("-F", type= str, default='bbc.parquet',
41. help= 'Name of file. It is assumed that it is located in the same location as this file')
42. input = parser.parse\_args()
43. main(input.N, input.Q, input.F, write=False)