COSC 6339

Fall 2018

Big Data Analytics

3rd Assignment

Jaivardhan Singh Shekhawat

Problem Description:

Part a:

Take your code from the second assignment and derive the solution for part2 and part3 into separate files/programs.

–Part 2 writes the inverted index into a textfile
–Part 3 reads the inverted index from the textfile

Spark with python language combines to become Pyspark. Apache Spark is an open-source distributed general-purpose cluster-computing framework. Originally developed at the University of California, Berkeley's AMPLab, the Spark codebase was later donated to the Apache Software Foundation, which has maintained it since. Cluster is nothing but a platform to install Spark. One can run Spark on distributed mode on the cluster. In the cluster, there is master and n number of workers. It schedules and divides resource in the host machine which forms the cluster. The prime work of the cluster manager is to divide resources across applications. It works as an external service for acquiring resources on the cluster.

- Part 2 writes the inverted index into a textfile

Solution Strategy:

First, we need to set application name, port number and executor memory. Then, we need to define spark context and spark session. We would be reading file, in which we kept top 1000 words, previously. Now, to remove punctuation, we used *string.punctuation*. We have defined a keyword *path* in which medium dataset is stored. *finalWords[]* is defined as array, as it will be used in storing the words from that file, and for that *for* loop is used. We need to convert file into rdd, so we converted and stored into *rdd* variable. Then, we need to swap the key with values and then punctuation can be removed

using *removePunct*. Then, we checked particular (from popular 1000 words) word is present or not using *checkWords()* function. Then, we need to convert the values of inverted index into decimal values, we used *float* function. Now, to add the values of word and file which has that word, we used *add* imported from *operator*. Since, word and file are key, and fraction is value, so we interchanged the form of that. We changed to word as a key and file(s) and fraction as values. Then, we grouped the key using *groupByKey()* function. Then map the values using *mapByValues()* function, passing *list* through it. Then, we need to map the values. Finally, to save the output in a file of hdfs cluster, we used *saveAsTextFile()* function.

- Part 3 reads the inverted index from the textfile

Solution Strategy:

First, we need to set application name, port number and executor memory. Then, we need to define spark context and spark session. Then, we need to define a variable which can store text file which we got from part 2 as a simple text file. Since, it is the simple text file, we need to map the key using *eval* function, then we need to map the key and value using regular lambda function. Now, we need to define a function named *func_similarity* passing variable, which takes the 2 documents and get the fraction of similarity between them by multiplying the fraction of similarity of both. Then, variable *sim_rdd* is defined to pass that function using *map* function. Then, we need to process key using *flatMap* function, to define new variable. After that we need to use *reduceByKey* function, using *add* operator and number of partitions to be 4. Then, before storing the data into file, we need to use *map* function to store into a new file. Finally, to save the output in a file of hdfs cluster, we used *saveAsTextFile()* function.

The execution time and size taken by code developed through this part is represented through tables and graphs in part e.

How to run code:

spark-submit --master yarn –num-executors 5 input_filename.py /cosc6339_hw2/gutenberg-500/ /bigd45/output_filename

How to see output:

hdfs dfs -cat /bigd45/output_filename/

Part b:

Implement a solution for Part 2 which writes the data using Avro, and Part 3 reads the inverted index as an Avro file and writes the final result as an Avro file

- Part 2 which writes the data using Avro

Solution Strategy:

First, we need to set application name, port number and executor memory. Then, we need to define spark context and spark session. As required, we need to read the file, so this program is reading the file and stored into *path1* variable. Also, we need to store medium dataset into a variable, we had done that using *path* variable. Now, to remove punctuation, we used *string.punctuation*. *finalWords[]* is defined as array, as it will be used in storing the words from that file, and for that *for* loop is used. We need to convert file into rdd, so we converted and stored into *rdd* variable. Then, we need to swap the key with values and then punctuation can be removed using *removePunct* through *filter* function. Then, we checked particular (from popular 1000 words) word is present or not using *checkWords()* function. Then, we need to convert the values of inverted index into decimal values, we used *float* function. Now, to add the values of word and file which has that word, we used *add* imported from *operator* and number of partitions to be 4.

Since, word and file are key, and fraction is value, so we interchanged the form of that. We changed to word as a key and file(s) and fraction as values. Then, we grouped the key using *groupByKey()* function. Then map the values using *mapByValues()* function, passing *list* through it. Then, we need to map the values. Now, before writing the data to *avro* format, we need to convert that data into dataframe, we had done that using *toDF* function by using schema. Finally, to save the output in a file of hdfs cluster in *avro* format, we used various functions such as *coalesce()* function to give the first non- null value among the given columns or null, *write* function to write the data, *format*() function to define the format in which that data needs to be stored and then *save()* function to save the data.

- Part 3 reads the inverted index as an Avro file and writes the final result as an Avro file

Solution Strategy:

First, we need to set application name, port number and executor memory. Then, we need to define spark context and spark session. As required, we need to read the file in the *avro* format, so we need to use *spark.read.format()* function to define the format, and *load()* function to load that specific *avro* file. Then, we need to convert the data to *rdd* format passing *list* as an argument. Now, we need to define a function named *func_similarity* passing variable, which takes the 2 documents and get the fraction of similarity between them by multiplying the fraction of similarity of both. Then, variable *sim_rdd* is defined to pass that function using *map* function. Then, we need to process key using *flatMap* function, to define new variable. After that we need to use *reduceByKey* function, using *add* operator and number of partitions to be 4. Then, before storing the data into file, we need to use *map* function to store into a new file. Now, before writing the data to *avro* format, we need to convert that data into dataframe, we had done that using *toDF* function by using schema. Finally, to save the output in a file of hdfs cluster in *avro* format, we used various functions such as *coalesce()* function to give the first non- null value among the given columns or null, *write* function to write the

data, *format*() function to define the format in which that data needs to be stored and then *save()* function to save the data.

The execution time and size taken by code developed through this part is represented through tables and graphs in part e.

How to run code:

spark-submit --master yarn –num-executors 5 input_filename.py /cosc6339_hw2/ gutenberg-500/ /bigd45/output_filename

How to see output:

hdfs dfs -cat /bigd45/output_filename/

Part c:

Same as part b, but using parquet files.

- Part 2 which writes the data using parquet

Solution Strategy:

First, we need to set application name, port number and executor memory. Then, we need to define spark context and spark session. As required, we need to read the file in parquet format, so this program is reading the file using *spark.read.load* function, so that particular parquet file is loaded and stored into *path1* variable. Also, we need to store medium dataset into a variable, we had done that using *path* variable. Now, to remove punctuation, we used *string.punctuation*. *finalWords[]* is defined as array, as it will be used in storing the words from that file, and for that *for* loop is used. We need to convert file into rdd, so we converted and stored into *rdd* variable. Then, we need to swap the key with values and then punctuation can be removed using

removePunct through *filter* function. Then, we checked particular (from popular 1000 words) word is present or not using *checkWords()* function. Then, we need to convert the values of inverted index into decimal values, we used *float* function. Now, to add the values of word and file which has that word, we used *add* imported from *operator* and number of partitions to be 4. Since, word and file are key, and fraction is value, so we interchanged the form of that. We changed to word as a key and file(s) and fraction as values. Then, we grouped the key using *groupByKey()* function. Then map the values using *mapByValues()* function, passing *list* through it. Then, we need to map the values. Now, before writing the data to *parquet* format, we need to convert that data into data frame, we had done that using *toDF* function by using passing schema. Finally, to save the output in a file of hdfs cluster in *parquet* format, we used various functions such as *coalesce()* function to give the first non- null value among the given columns or null, *write* function to write the data, *option()* function is used for non-compression of data by setting *"compression"* value as *"none"* and then *parquet()* function to save the data in parquet format.

- Part 3 reads the inverted index as an Parquet file and writes the final result as an Parquet file

Solution Strategy:

First, we need to set application name, port number and executor memory. Then, we need to define spark context and spark session. As required, we need to read the file in parquet format, so this program is reading the file using *spark.read.load* function, so that particular parquet file is loaded and stored into *inverted8* variable. Then, we need to convert the data to *rdd* format passing *list* as an argument. Now, we need to define a function named *func_similarity* passing variable, which takes the 2 documents and get the fraction of similarity between them by multiplying the fraction of similarity of both. Then, variable *sim_rdd* is defined to pass that function using *map* function. Then, we need to process key using *flatMap* function, to define new variable. After that we need to use *reduceByKey* function, using *add* operator

and number of partitions to be 4. Then, before storing the data into file, we need to use *map* function to store into a new file. Now, before writing the data to *parquet* format, we need to convert that data into data frame, we had done that using *toDF* function by using passing schema. Finally, to save the output in a file of hdfs cluster in *parquet* format, we used various functions such as *coalesce()* function to give the first non- null value among the given columns or null, *write* function to write the data, *option()* function is used for non-compression of data by setting *"compression"* value as *"none"* and then *parquet()* function to save the data in parquet format.

The execution time and size taken by code developed through this part is represented through tables and graphs in part e.

How to run code:

spark-submit --master yarn –num-executors 5 input_filename.py /cosc6339_hw2/ gutenberg-500/ /bigd45/output_filename

How to see output:

hdfs dfs -cat /bigd45/output_filename/

Part d:

Implement a version of either Part b or Part c (its your chose) that creates snappy-compressed files (either Avro or Parquet).

- Implementing version of part c that creates snappy compressed files using parquet

Solution Strategy:

First, we need to set application name, port number and executor memory. Then, we need to define spark context and spark session. As required, we need

to read the file in parquet format, so this program is reading the file using *spark.read.load* function, so that particular parquet file is loaded and stored into *inverted8* variable. Then, we need to convert the data to *rdd* format passing *list* as an argument. Now, we need to define a function named *func_similarity* passing variable, which takes the 2 documents and get the fraction of similarity between them by multiplying the fraction of similarity of both. Then, variable *sim_rdd* is defined to pass that function using *map* function. Then, we need to process key using *flatMap* function, to define new variable. After that we need to use *reduceByKey* function, using *add* operator and number of partitions to be 4. Then, before storing the data into file, we need to use *map* function to store into a new file. Now, before writing the data to *parquet* format, we need to convert that data into data frame, we had done that using *toDF* function by using passing schema. Finally, to save the output in a file of hdfs cluster in *parquet* format, we used various functions such as *coalesce()* function to give the first non- null value among the given columns or null, *write* function to write the data, *option()* function is used for compression of data by setting *"compression"* value as *"snappy"* and then *parquet()* function to save the data in parquet format.

The execution time and size taken by code developed through this part is represented through tables and graphs in part e.

How to run code:

spark-submit --master yarn –num-executors 5 input_filename.py /cosc6339_hw2/gutenberg-500/ /bigd45/output_filename

How to see output:

hdfs dfs -cat /bigd45/output_filename/

Part e:

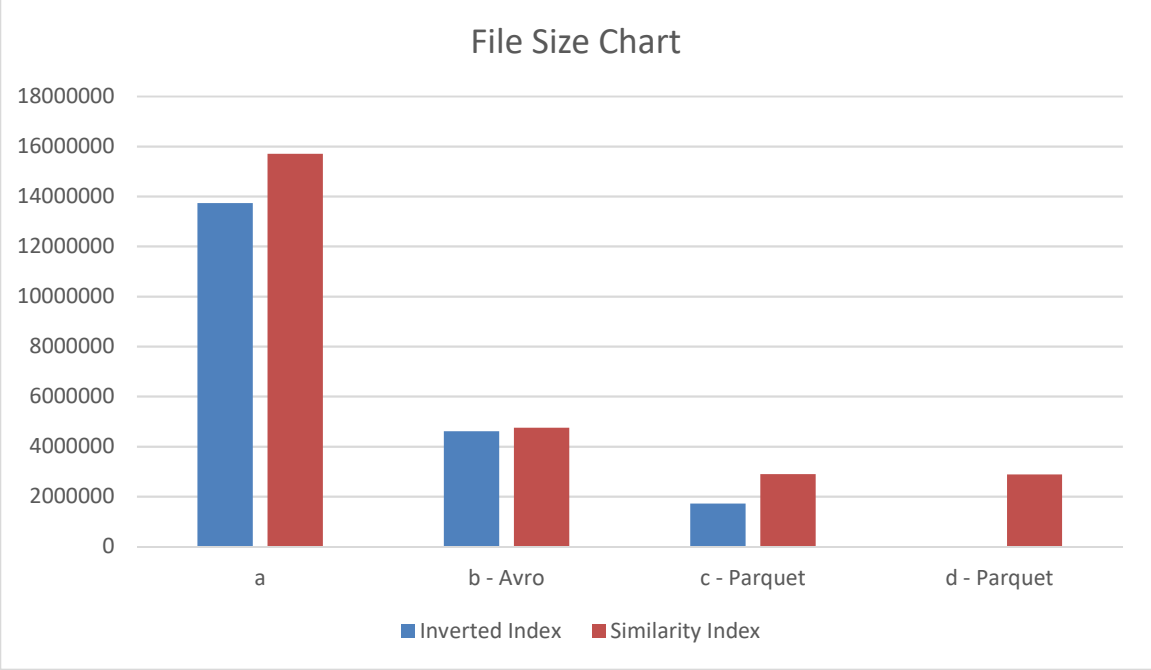
Compare file sizes for Parts a-d, as well as execution time of all versions of Part a-d using the corresponding format using the MEDIUM data set using 5 executors.

File sizes for parts a-d using MEDIUM dataset with 5 executors are represented in the following table.

Table 1.1: File sizes for parts a-d:

Part	Format	Size (in bytes)
a	Inverted Index	13725698
	Similarity Matrix	15697356
b – Avro	Inverted Index	4610087
	Similarity Matrix	4756637
c – Parquet	Inverted Index	1721180
	Similarity Matrix	2896663
d – Parquet	Similarity Matrix	2881431

File sizes are represented in the following graph.



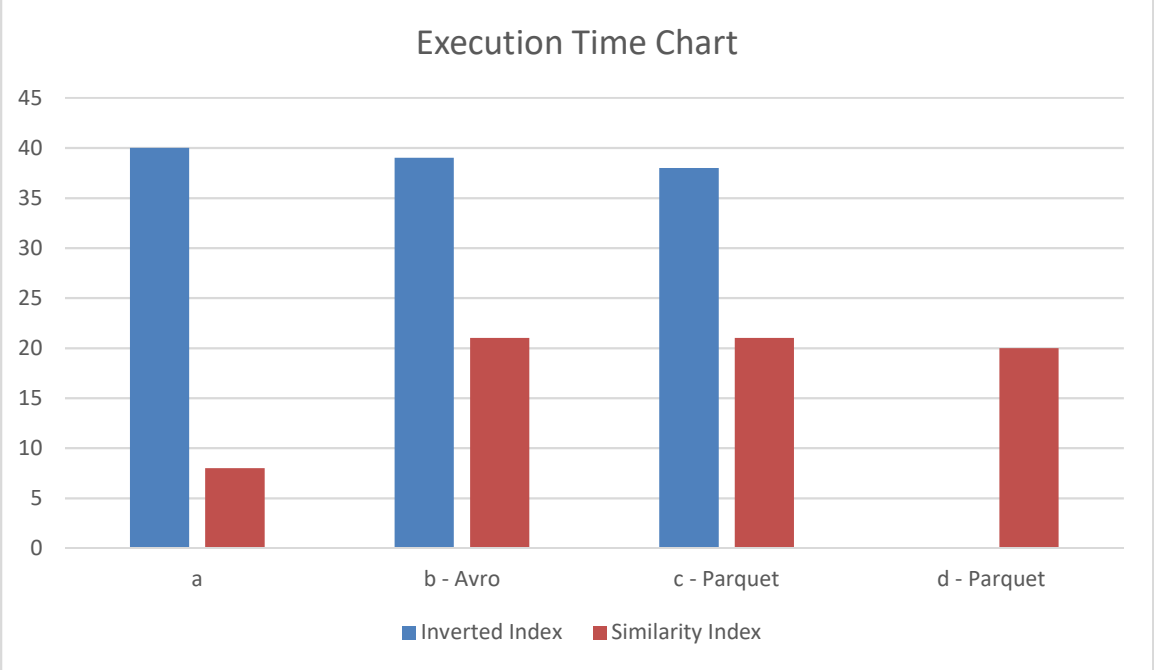
Graph 1.1: File size chart for parts a-d

Execution time of all versions of part a-d using MEDIUM dataset with 5 executors are represented in the following table.

Table 1.2: Average execution time for parts a-d:

Part	Format	Avg. Time (in mins.)
a	Inverted Index	40
	Similarity Matrix	8
b – Avro	Inverted Index	39
	Similarity Matrix	21
c – Parquet	Inverted Index	38
	Similarity Matrix	21
d - Parquet	Similarity Matrix	20

Execution time are represented in the following graph.



Graph 1.2: Execution time chart for parts a-d

Resources Used:

Cluster:

50 Appro 1522H nodes (whale-001 to whale-057), each node with

- two 2.2 GHz quad-core AMD Opteron processor (8 cores total)
- 16 GB main memory
- Gigabit Ehternet
- 4xDDR InfiniBand HCAs (not used at the moment)

Network Interconnect

- 144 port 4xInfiniBand DDR Voltaire Grid Director ISR 2012 switch (donation from TOTAL)
- two 48 port HP GE switch

Storage

- 4 TB NFS /home file system (shared with crill)
- ~7 TB HDFS file system (using triple replication)

Analysis:

After execution of code in cluster and from tables and graph, I came to final analysis according to execution time taken and file size generated. If we are going to talk regarding file size then, we can easily say that without using Avro or Parquet file format, going with simple text file, the file size generated will be larger. If we are going to implement inverted index and similarity matrix using Avro or parquet, file size generated will be small as compared to simple text file. In part d, we had used parquet to snappy-compress the file and file size is lesser. Going to next parameter, i.e. execution time taken by the code developed through all parts described above, we can easily say that writing the data into text file in inverted index part, time taken is much more as compared to reading the data from the text file in similarity index part. If we compare according to file format, i.e. Avro and Parquet, time taken by parquet file format is less than avro file format. And, if we compare the time taken according to compression then, snappy compressed file (using parquet) has took less time than non-compressed file.