COSC 6339
Big Data Analytics
Homework – 3

Priscilla Imandi
1619570

Problem Description

An extension of HW-2, where the goal was to find the similarity between 2 large documents.
The goal is to

i) Modularize the previous work into
  - Finding an Inverted index of the popular words
  - Calculating the similarity matrix from the Inverted index

  Using the text file format

ii) Implementing the same for Avro data format in an uncompressed version

iii) Implementing the same for parquet data format in an uncompressed version

iv) Using snappy compression

v) Performing a comparative study on the file sizes and execution times

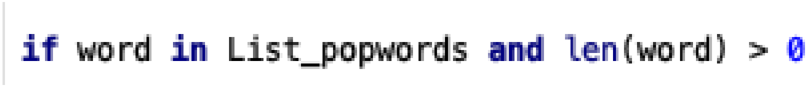
Solution Explanation

i) Modularizing for text files

I have already completed this step in the previous homework. The list of popular words can be used from the file previously generated or by re-generating again – a list of 1000 most popular words.

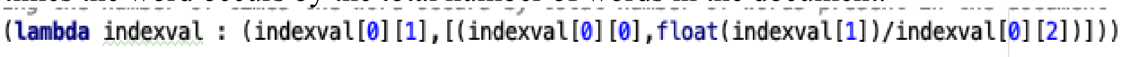
Creating the Inverted Index

- **Collecting Popular Words**
  Read the output files from part-1, split them based on the space delimiter and add the words to a list
- Using the same process as part-1, read the entire folder of the input dataset and pre-process it. And check if the word is present in the list of popular words.



```
if word in List_popwords and len(word) > 0
```

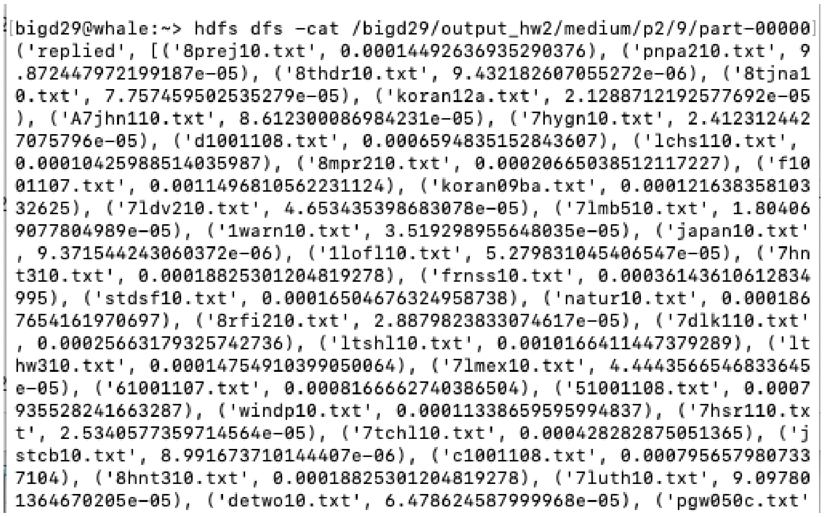
- Apply the reduce operation and get the word count. Now map each word and document to the corresponding weight. This is done by dividing the number of times the word occurs by the total number of words in the document.



```
(lambda indexval : (indexval[0][1],[(indexval[0][0],float(indexval[1])/indexval[0][2])]))
```

  Word weight = word count / total number of words in document
- Reduce by add key – Now the key would be the word and the value would be a list of tuples of all the documents and their respective weights.

Output File:

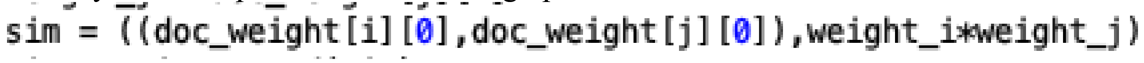


```
bigd29@whale:~> hdfs dfs -cat /bigd29/output_hw2/medium/p2/9/part-00000
('replied', [('8prej10.txt', 0.00014492636935290376), ('pnpa210.txt', 9
.872447972199187e-05), ('8thdr10.txt', 9.432182607055272e-06), ('8tjna1
0.txt', 7.757459502535279e-05), ('koran12a.txt', 2.1288712192577692e-05
), ('A7jhn110.txt', 8.612300086984231e-05), ('7hygn10.txt', 2.412312442
7075796e-05), ('d1001108.txt', 0.0006594835152843607), ('lchs110.txt',
0.00010425988514035987), ('8mpr210.txt', 0.00020665038512117227), ('f10
01107.txt', 0.0011496810562231124), ('koran09ba.txt', 0.00012163835810З
32625), ('7ldv210.txt', 4.653435398683078e-05), ('7lmb510.txt', 1.80406
9077804989e-05), ('1warn10.txt', 3.519298955648035e-05), ('japan10.txt'
, 9.371544243060372e-06), ('1lofl10.txt', 5.279831045406547e-05), ('7hn
t310.txt', 0.00018825301204819278), ('frnss10.txt', 0.00036143610612834
995), ('stdsf10.txt', 0.00016504676324958738), ('natur10.txt', 0.000186
7654161970697), ('8rfi210.txt', 2.8879823833074617e-05), ('7dlk110.txt'
, 0.00025663179325742736), ('ltshl10.txt', 0.0010166411447379289), ('lt
hw310.txt', 0.00014754910399050064), ('7lmex10.txt', 4.4443566546833645
e-05), ('61001107.txt', 0.0008166662740386504), ('51001108.txt', 0.0007
935528241663287), ('windp10.txt', 0.00011338659595994837), ('7hsr110.tx
t', 2.5340577359714564e-05), ('7tchl10.txt', 0.000428282875051365), ('j
stcb10.txt', 8.991673710144407e-06), ('c1001108.txt', 0.000795657980733
7104), ('8hnt310.txt', 0.00018825301204819278), ('7luth10.txt', 9.09780
1364670205e-05), ('detwo10.txt', 6.478624587999968e-05), ('pgw050c.txt'
```

- A part of the output file can be seen above.
- In the above picture – ‘replied’ is the key. And the values following it are a list of tuples of document and the respective weight of ‘replied’

Finding the similarity Matrix

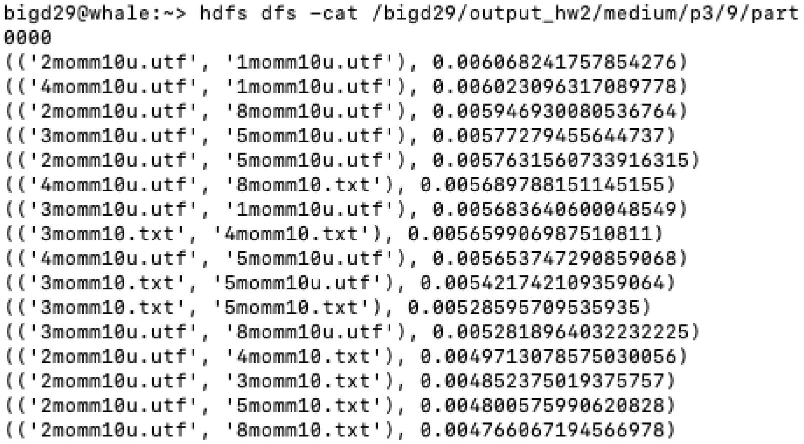
- Collecting Inverted Index Values
  Read the output files from part-2, split them based on a \n character and store them in a RDD
- Now after processing we have the data in the processable format but data is same as output generated by part-2.
- The converted list also called as postings has 2 values for every row – At index 0 it has the word and at index 1 it has the list of tuples of documents and their respective weights.
- For every word we perform the following operation

  ```
  sim = ((doc_weight[i][0],doc_weight[j][0]),weight_i*weight_j)
  ```

  In the above line, doc_weight datastructure contains all list of tuples of documents and their respective weights.
  Now at index 0, i.e doc_weight[x]**[0]**, we have the names of the documents. And the index 1 has weights. weight_i corresponds to weight of document i and weight_j to document j. Multiplying these weights would give the similarity between those documents
- Now we have a matrix of document tuples and their corresponding weights for various words(words are not written into the RDD).
- Now, perform a reduce operation, this would add all the similarity values of various words for all the pair of documents.
- This would be our Similarity index

Output File:



```
bigd29@whale:~> hdfs dfs -cat /bigd29/output_hw2/medium/p3/9/part
0000
(('2momm10u.utf', '1momm10u.utf'), 0.006068241757854276)
(('4momm10u.utf', '1momm10u.utf'), 0.006023096317089778)
(('2momm10u.utf', '8momm10u.utf'), 0.005946930080536764)
(('3momm10u.utf', '5momm10u.utf'), 0.0057727945564737)
(('2momm10u.utf', '5momm10u.utf'), 0.005763156073391631 5)
(('4momm10u.utf', '8momm10.txt'), 0.005689788151145155)
(('3momm10u.utf', '1momm10u.utf'), 0.005683640600048549)
(('3momm10.txt', '4momm10.txt'), 0.005659906987510811)
(('4momm10u.utf', '5momm10u.utf'), 0.005653747290859068)
(('3momm10.txt', '5momm10u.utf'), 0.005421742109359064)
(('3momm10.txt', '5momm10.txt'), 0.00528595709535935)
(('3momm10u.utf', '8momm10u.utf'), 0.0052818964032232225)
(('2momm10u.utf', '4momm10.txt'), 0.004971307857503005 6)
(('2momm10u.utf', '3momm10.txt'), 0.004852375019375757)
(('2momm10u.utf', '5momm10.txt'), 0.004800575990620828)
(('2momm10u.utf', '8momm10.txt'), 0.004766067194566978)
```

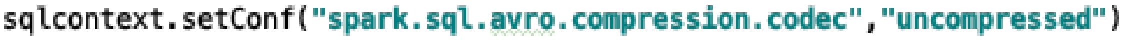
- It can be seen that the document pair(tuple) and the similarity weight are seen.
- Even here, the files for the bigger dataset will be larger when compared to the medium dataset because the pairs(combinations) of documents will be more in the bigger dataset.

ii) Implementation of Avro Uncompressed Version

Avro stores the data definition in JSON format making it easy to read and interpret, the data itself is stored in binary format making it compact and efficient. Avro files include markers that can be used to splitting large data sets into subsets.

It is saved with the .avro extension and can only be opened be opened in a readable format after converting it to a data frame.
The following steps have to be added to the solution code

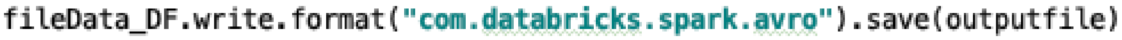
- Setting the SQL context to an uncompressed format has to be done

  ```
  sqlcontext.setConf("spark.sql.avro.compression.codec","uncompressed")
  ```

  Avro compression can be set to uncompressed, snappy and deflate. The levels for deflating can be specified.

- Writing the file as an avro file
  A dataframe can be written into avro format using the following command

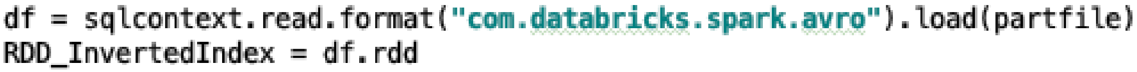
  

  ```
  fileData_DF.write.format("com.databricks.spark.avro").save(outputfile)
  ```

  The “outputfile” is the directory where the avro file has to be saved

  The above line has to be specified in the place of saving to a text file

- Reading an avro file

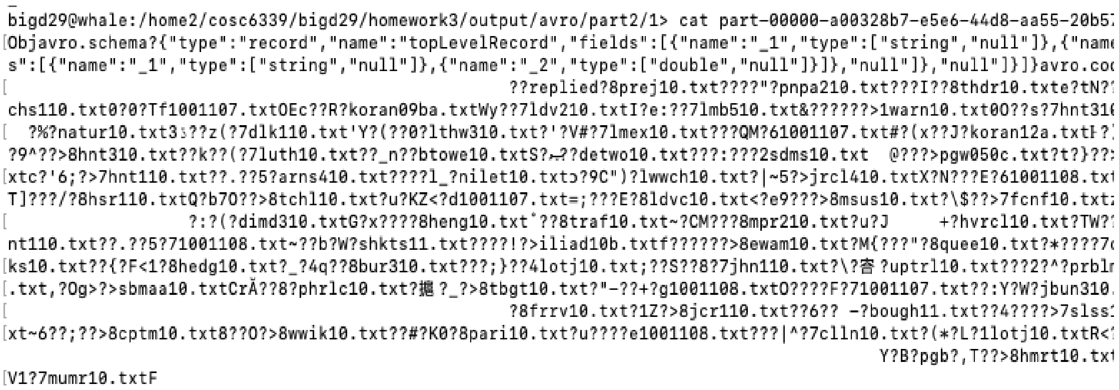
  ```
  df = sqlcontext.read.format("com.databricks.spark.avro").load(partfile)
  RDD_InvertedIndex = df.rdd
  ```

  The avro file can be loaded using the load command

The “partfile” is the directory where all the avro files are located.
The avro file has to be converted to an RDD before performing any operations on it.
This additional step has to be added in the place of reading the text files.

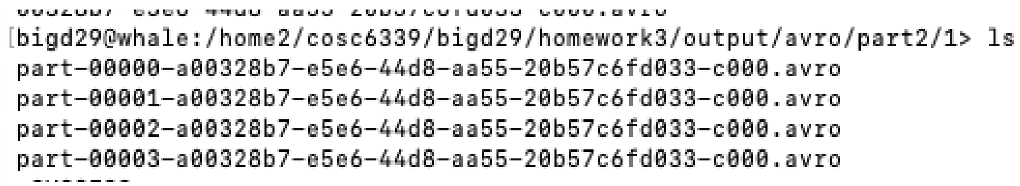
Output File:

If we try to read an Avro file, it shows all gibberish along with a header metadata.



```
bigd29@whale:/home2/cosc6339/bigd29/homework3/output/avro/part2/1> cat part-00000-a00328b7-e5e6-44d8-aa55-20b5
Objavro.schema?{"type":"record","name":"topLevelRecord","fields":[{"name":"_1","type":["string","null"]},{"nam
s":[{"name":"_1","type":["string","null"]},{"name":"_2","type":["double","null"]}]},"null"]},"null"]}]}avro.co
                                                      ??replied?8prej10.txt????"?pnpa210.txt???I??8thdr10.txte?tN?
chs110.txt0?0?Tf1001107.txtOEc??R?koran09ba.txtWy??7ldv210.txtI?e:??7lmb510.txt&??????>1warn10.txt0O??s?7hnt31
  ?%?natur10.txt3:??z(?7dlk110.txt'Y?(??0?lthw310.txt?'?V#?7lmex10.txt???QM?61001107.txt#?(x??J?koran12a.txt?
?9^??>8hnt310.txt??k??(?7luth10.txt??_n??btowe10.txtS?⤶??detwo10.txt???:???2sdms10.txt  @???>pgw050c.txt?t?}??
xtc?'6;?>7hnt110.txt??.??5?arns410.txt????l_?nilet10.txtↄ?9C")?lwwch10.txt?|~5?>jrcl410.txtX?N???E?61001108.tx
T]???/?8hsr110.txtQ?b7O??>8tchl10.txt?u?KZ<?d1001107.txt=;???E?8ldvc10.txt<?e9???>8msus10.txt?\$??>7fcnf10.txt
                    ?:?(?dimd310.txtG?x????8heng10.txt˚??8traf10.txt~?CM???8mpr210.txt?u?J     +?hvrcl10.txt?TW?
nt110.txt??.??5?71001108.txt~??b?W?shkts11.txt????!?>iliad10b.txtf??????>8ewam10.txt?M{???"?8quee10.txt?*????7
ks10.txt??{?F<1?8hedg10.txt?_?4q??8bur310.txt???;}??4lotj10.txt;??S??8?7jhn110.txt?\?客?uptrl10.txt???2?^?prbl
.txt,?Og>?>sbmaa10.txtCrĂ??8?phrlc10.txt?摠?_?>8tbgt10.txt?"-??+?g1001108.txtO????F?71001107.txt??:Y?W?jbun310
                                                      ?8frrv10.txt?1Z?>8jcr110.txt??6?? -?bough11.txt??4????>7slss
xt~6??;??>8cptm10.txt8??O?>8wwik10.txt??#?K0?8pari10.txt?u????e1001108.txt???|^?7clln10.txt?(*?L?1lotj10.txtR<
                                                                                          Y?B?pgb?,T??>8hmrt10.tx
V1?7mumr10.txtF
```

The output file has a .avro extension.
Even if the compression format is changed the extension remains the same.



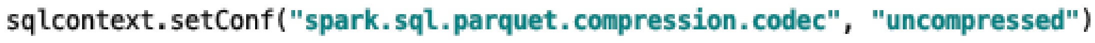
```
bigd29@whale:/home2/cosc6339/bigd29/homework3/output/avro/part2/1> ls
part-00000-a00328b7-e5e6-44d8-aa55-20b57c6fd033-c000.avro
part-00001-a00328b7-e5e6-44d8-aa55-20b57c6fd033-c000.avro
part-00002-a00328b7-e5e6-44d8-aa55-20b57c6fd033-c000.avro
part-00003-a00328b7-e5e6-44d8-aa55-20b57c6fd033-c000.avro
```

iii) Implementation of Parquet Uncompressed Version

Apache Parquet. Apache Parquet is a columnar storage format available to any project in the Hadoop ecosystem, regardless of the choice of data processing framework, data model or programming language.
Apache Parquet is implemented using the record-shredding and assembly algorithm, which accommodates the complex data structures that can be used to store the data. The values in each column are physically stored in contiguous memory locations.
Avro is row-based and parquet is column-based storage.

The following steps have to be added to the solution code

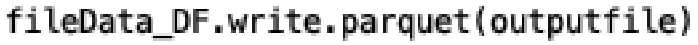
- Setting the SQL context to an uncompressed format has to be done



```
sqlcontext.setConf("spark.sql.parquet.compression.codec", "uncompressed")
```

  Parquet compression can be set to none, snappy, gzip, izo.

- Writing the file as a parquet file

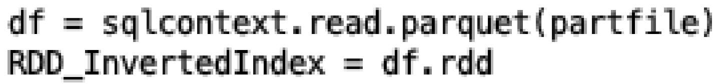


```
fileData_DF.write.parquet(outputfile)
```

The "outputfile" is the directory where the parquet file has to be saved

The above line has to be specified in the place of saving to a text file

- Reading a parquet file

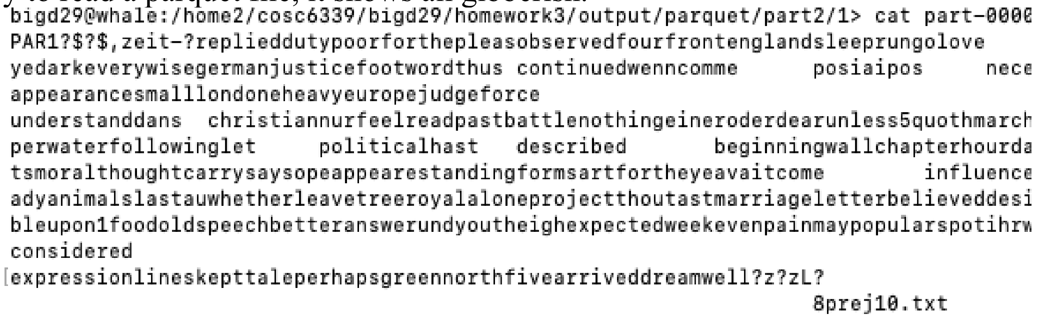


```
df = sqlcontext.read.parquet(partfile)
RDD_InvertedIndex = df.rdd
```

The parquet file can be loaded using the read command
The "partfile" is the directory where all the parquet files are located.
The parquet file has to be converted to an RDD before performing any operations on it.
This additional step has to be added in the place of reading the text files.

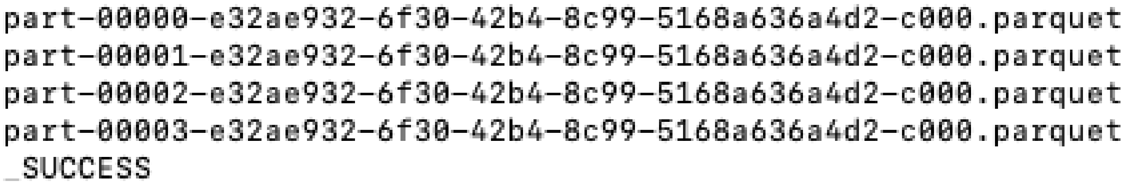
Output File:

If we try to read a parquet file, it shows all gibberish.



```
bigd29@whale:/home2/cosc6339/bigd29/homework3/output/parquet/part2/1> cat part-000
PAR1?$?$,zeit-?replieddutypoorforthepleasobservedfourfrontenglandsleeprungolove
yedarkeverywisegermanjusticefootwordthus continuedwenncomme      posiaipos     nece
appearancesmalllondoneheavyeuropejudgeforce
understanddans  christiannurfeelreadpastbattlenothingeineroderdearunless5quothmarch
perwaterfollowinglet     politicalhast   described       beginningwallchapterhourda
tsmoralthoughtcarrysaysopeappearestandingformsartfortheyeavaitcome       influence
adyanimalslastauwhetherleavetreeroyalaloneprojectthoutastmarriageletterbelieveddesi
bleupon1foodoldspeechbetteranswerundyoutheighexpectedweekevenpainmaypopularspotihrw
considered
[expressionlineskepttaleperhapsgreennorthfivearriveddreamwell?z?zL?
                                                               8prej10.txt
```

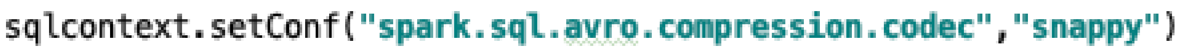
It is saved with a .parquet extension.



```
part-00000-e32ae932-6f30-42b4-8c99-5168a636a4d2-c000.parquet
part-00001-e32ae932-6f30-42b4-8c99-5168a636a4d2-c000.parquet
part-00002-e32ae932-6f30-42b4-8c99-5168a636a4d2-c000.parquet
part-00003-e32ae932-6f30-42b4-8c99-5168a636a4d2-c000.parquet
_SUCCESS
```

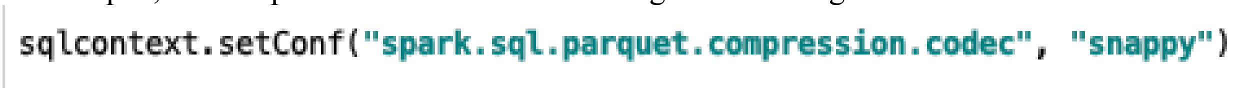
iv) **Implementation of Avro & Parquet Snappy Version**

For avro, the compression can be set to set using the following line.



```
sqlcontext.setConf("spark.sql.avro.compression.codec","snappy")
```

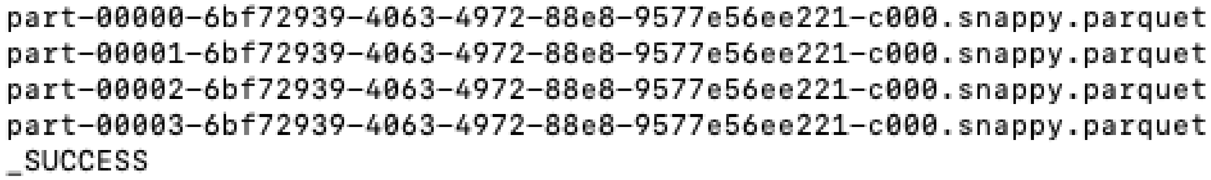
For Parquet, the compression can be set to set using the following line



```
sqlcontext.setConf("spark.sql.parquet.compression.codec", "snappy")
```

Output File:

Snappy compressed version of Avro is same as uncompressed but for parquet files the extension changes.
The extension is now snappy.parquet



```
part-00000-6bf72939-4063-4972-88e8-9577e56ee221-c000.snappy.parquet
part-00001-6bf72939-4063-4972-88e8-9577e56ee221-c000.snappy.parquet
part-00002-6bf72939-4063-4972-88e8-9577e56ee221-c000.snappy.parquet
part-00003-6bf72939-4063-4972-88e8-9577e56ee221-c000.snappy.parquet
_SUCCESS
```

Challenges and Findings:

i) For the normal text version, it the same process – but when implementing the avro and parquet versions – it is a little tricky because we cannot view the contents of the intermediate files to ensure that the data is correct.
ii) Only after completing the entire process and reaching the final output of similar documents we can verify.
iii) For the avro and parquet versions, just changing the way the file was read and written was not sufficient.
iv) On trying to pre-process the RDD generated after loading the avro and parquet files had issues as the individual contents of the RDD were considered as lists.
v) So directly performing the required function – like finding the similarity matrix was a solution to the issue, rather than pre-processing it first.

Specifications of the System:

Whale Cluster :

i) Nodes:
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 Ethernet
- 4xDDR InfiniBand HCAs (not used at the moment)

ii) Network Interconnect

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

iii) Storage

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

Spark in Python:

i) It is a unified analytics engine for big data processing, with built-in modules for streaming, SQL, machine learning and graph processing

ii) It can be used with Scala, Python, R and SQL shells.

iii) Spark 2.2.2

iv) Hadoop 3.0.3

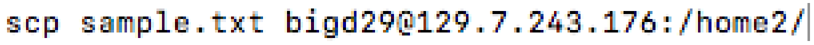
v) Open JDK 64-Bit Server VM, 1.8.0_171

Execution of the code

i) Copying the source code file from the local Desktop to Remotely connected Linux system.

```
Scp <local-location>   Username@Host: <Remote-Location>
```

Enter the password when prompted for it



```
scp sample.txt bigd29@129.7.243.176:/home2/
```

ii) Submitting a job:

```
spark-submit --master yarn --conf spark.ui.port=4060 --num-executors 5 --executor-cores 4 --driver-cores 4 part2_avro.py /cosc6339_hw2/gutenberg-500/ /bigd29/output_hw3/part2/avro/1
```

The above command is used to run the job, which indicates the number of executors used, the number of executor-cores that can be allotted, name of the source file, hdfs location of the data file, hdfs location of the output directory

iii) Viewing the output:

```
hdfs dfs -cat /bigd29/output_hw3/ part4/avro/1/part-00000
```

The above command has the location of the output file.

iv) Getting the data from Hadoop to Local file system:
This has to be done to find the size of the intermediate avro/parquet/text files.

```
hdfs dfs -get /bigd29/output_hw3/part2/avro/15 /home2/cosc6339/bigd29/homework3/output/avro_snappy/part2/
```

v) Finding the size of the file
The size of the intermediary files in a human readable format

```
du -h <filename>
```

Results & Finding:

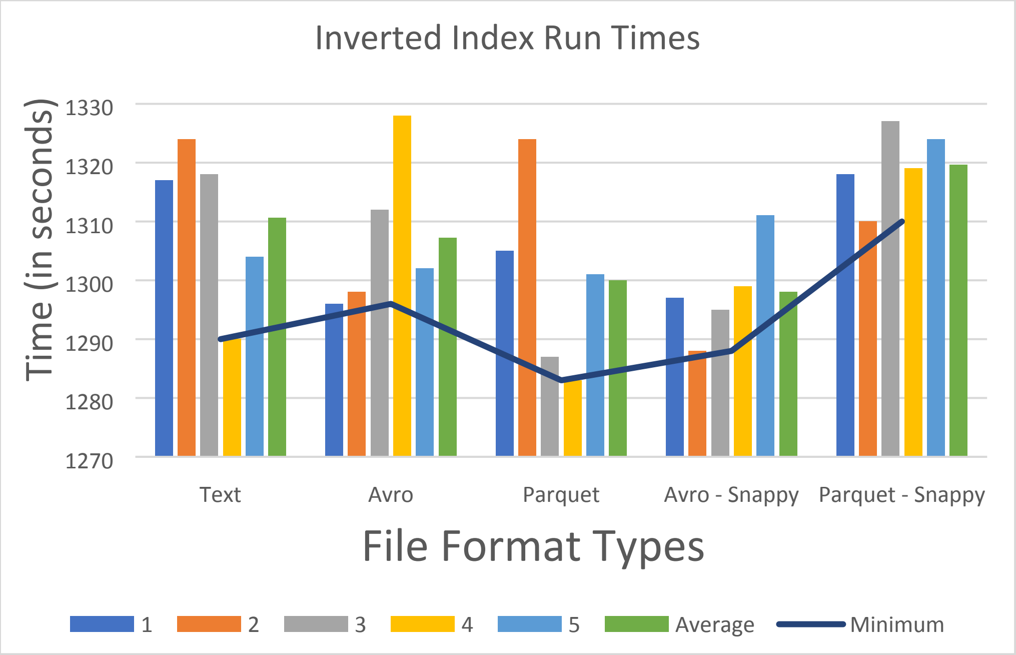
Run Time of the Solution Files

i) Inverted Index

Runs\File formats	Text (in sec)	Avro (in sec)	Parquet (in sec)	Avro - Snappy (in sec)	Parquet - Snappy (in sec)
1	1317	1296	1305	1297	1318
2	1324	1298	1324	1288	1310
3	1318	1312	1287	1295	1327
4	1290	1328	1283	1299	1319
5	1304	1302	1301	1311	1324
Average	1310.6	1307.2	1300	1298	1319.6
Minimum	1290	1296	1283	1288	1310

The above table depicts the time in which the solution runs for the various file types.

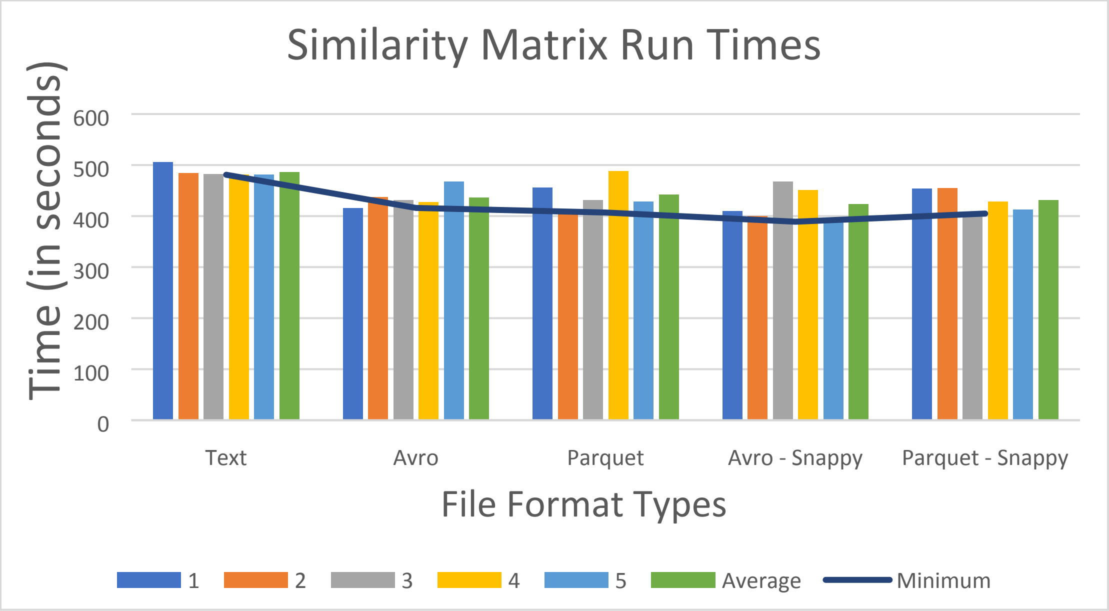
- We can see that the minimum time is taken by Parquet files which is obvious.
- It is unexpected to see that Parquet – Snappy files take longer to run. This might be due to the additional compression. While reading or writing a compressed file, additional computations have to be done. I am assuming these additional operations are the reason for the longer duration.
- Snappy is faster when compared to other compression methods like zlib but the resultant file is larger.
- Another reason for parquet being faster might also be due to the data we are using – if the schema is too nested, this would result in parquet files taking longer time. The task at hand also influences the time taken.
- Hence, the trend goes in this way – parquet being the fastest, then avro/ avro-snappy, text and finally parquet-snappy. The main reason for parquet snappy being the slowest is due to the compression ratio.



ii) Similarity Matrix

Runs\File formats	Text (in sec)	Avro (in sec)	Parquet (in sec)	Avro - Snappy (in sec)	Parquet - Snappy (in sec)
1	506	416	456	410	454
2	484	437	407	400	455
3	482	431	431	468	405
4	481	428	488	451	429
5	481	468	429	389	413
Average	486.8	436	442.2	423.6	431.2
Minimum	481	416	407	389	405

- In the previous task, there was no read operation. We had a list of words from which we finally got the avro/parquet files and wrote them down onto the disk.
- This difference of **Read** operation is the main reason for the change in trend.
- Paquet needs additional parsing to format the data increasing the read time
- We can see that Avro- snappy compressed takes the least amount of time, followed avro in the uncompressed form. Avro has an edge over other formats when read operation is performed.



- This can be due to querying – Avro is good with row selection whereas parquet with columns. Since there are not too many column-oriented structures avro has performed better.
- The trend is avro/avro-snappy followed by parquet/parquet snappy and finally text. The average time is considered to verify the trend.

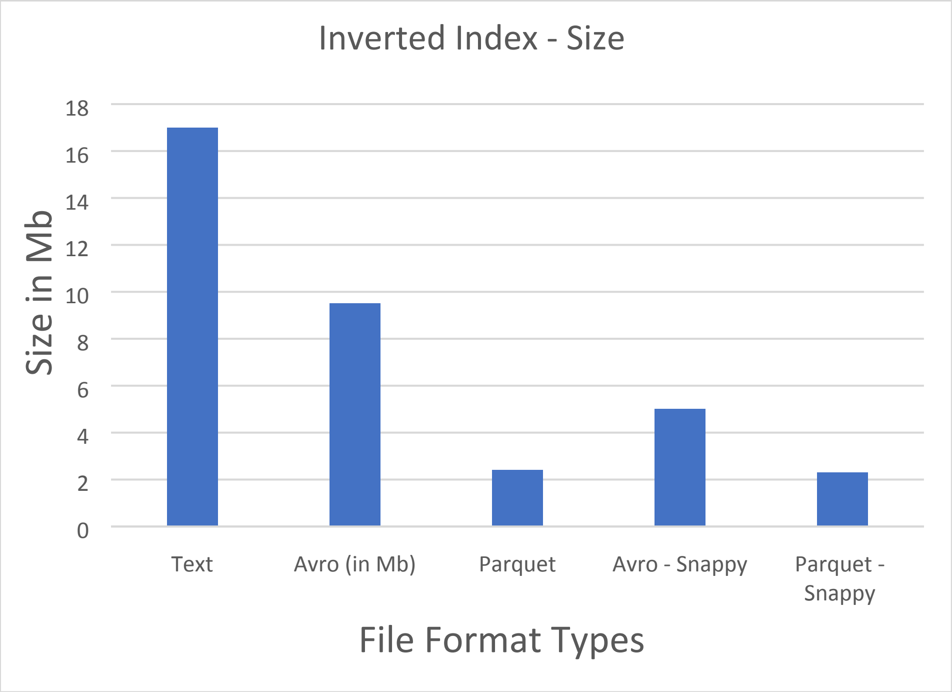
- We can also see that unlike part-2 (no read), here the text file solution was way worse than parquet-snappy.

Size of Intermediary Files Generated

i) Inverted Index

Runs\File formats	Text (in Mb)	Avro (in Mb)	Parquet (in Mb)	Avro - Snappy (in Mb)	Parquet - Snappy (in Mb)
1	17	9.5	2.4	4.9	2.3
2	17	9.5	2.4	4.9	2.3
3	17	9.5	2.4	5.2	2.3
4	17	9.5	2.4	5.2	2.3
5	17	9.5	2.4	4.9	2.3
Average	17	9.5	2.4	5.02	2.3
Minimum	17	9.5	2.4	4.9	2.3

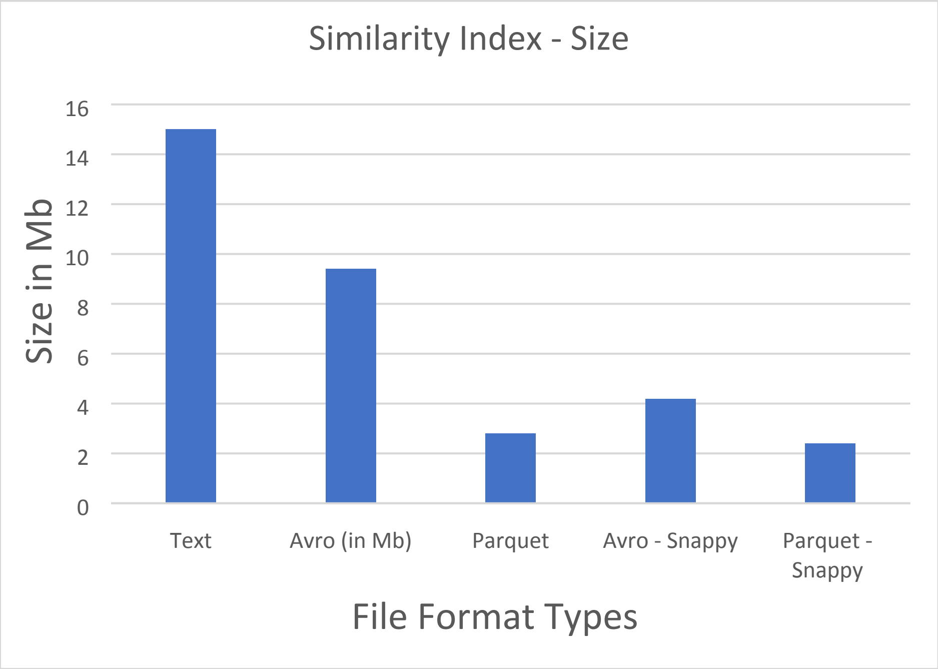
- Paquet files are the smallest of the files – this is due to hybrid compression methods used.
- Parquet implements a hybrid of bit packing and RLE(run length encoding), in which the encoding switches based on which produces the best compression results.
- For small integers, packing multiple integers into the same space makes storage more efficient – used by parquet.
- The difference between snappy parquet and uncompressed avro files is not much because the files have already been compressed.
- Text files are having a human readable format and hence the large size, whereas avro and parquet are encoded.
- In text files – every line is a record and has to be parsed and hence the huge size.
- We can see that snappy compressed avro file is almost half the size of an avro file – because avro files are basically json files – an avro container consists of a header followed by a few blocks
- Another important concept is file spilittability – ability to read a part of the file rather than the entire file. Avro and parquet offer this due to their compressed binary nature and hence the size.
- Parquet files are clearly the winner followed by avro and then the plain text file.



ii) Similarity Matrix

Runs\File formats	Text (in Mb)	Avro (in Mb)	Parquet (in Mb)	Avro - Snappy (in Mb)	Parquet - Snappy (in Mb)
1	15	9.4	2.8	4.1	2.8
2	15	9.4	2.8	4.3	2.3
3	15	9.4	2.8	4.1	2.3
4	15	9.4	2.8	4.1	2.3
5	15	9.4	2.8	4.3	2.3
Average	15	9.4	2.8	4.18	2.4
Minimum	15	9.4	2.8	4.1	2.3

- The trend in file size is similar to part-2 but the exact numbers differ a little. This might be due to the data schema.
- Inverted index consisted of each word and a list of files and the corresponding weights, where as now the schema is different – we just have a key value pair of the pair of documents and their corresponding similarity.
- Part-3 schema is less complex when compared to part-2 and this may be the reason for the change in differences.



Conclusion

From the above results and findings, we can conclude that parquet files are much better than other formats.
If disk space and complex querying is the first priority then parquet is definitely the winner.
Avro is a Row based format. If you want to retrieve the data as a whole you can use Avro.
Parquet is a Column based format. If your data consists of lot of columns but you are interested in a subset of columns then you can use Parquet.

Avro, being a row-based file format, is best fit for write intensive operation, which is clearly seen in part-3. Using the avro in its snappy compressed form is better for read, but parquet is much better if it is just write. Also, a lot of space can be saved using the parquet files.

Using snappy compression – further reduces the disk space used. It is a good compressor as well as a good decompressor.

In part-3 decompression has to be done on the already compressed files to retrieve the data. In part-2 avro snappy did way better than parquet snappy, but in part 3 it is almost similar to parquet-snappy. Hence, we can make a conclusion that avro-snappy is a much better compressor than decompressor.