Big Data Analytics Homework 3- Final Report

Avro and Parquet and Snappy Compressed File

Lavanya.S.S[16161056]
NSM, Computer science

Problem Description – Part 1:

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

Input files

Medium input (500 short documents) available in hdfs in "/cosc6339_hw2/gutenberg-500/"

The medium dataset contains 500 documents. Which consist of various text files each file having certain amount of text data.

Solution Strategy – Part 1

1. Initially we will have 1000 most occurring words from the dataset which was executed as part of homework 2. The word list will be saved under the directory "/bigd27/thousandwordsniv" , Which is the starting point for creating the inverted index.
2. Inverted index is a structure which will have the words and weight of each word in each corresponding document.
3. Similarity matrix is calculated based on the formula "S(docx, docy) = $t \epsilon V(weightt\_docx \times weightt\_docy)$" , which will eventually find out which pair of documents are similar to each other based on most frequently occurring words.
4. So in the part A of HW3, we are dividing the solution of hw2 into two parts, in the first part the wordlist is read and inverted index is written as a text file, this program is saved in the name - 'Ainvertedindex.py'
5. The same inverted index text file is read by the third part and similarity matrix is created which is again stored in a directory. The program is saved in the name 'ASimilaritymatrix.py'
6. The command to execute these files are as shown below

   Spark-submit –master yarn –num-executors 5 AInvertedindex.py

   Spark-submit –master yarn –num-executors 5 ASimilaritymatrix.py

   The output is stored under the directory /bigd27/Inverted_IndexMediumdatasetFinal, whose execution took around 24.9 minutes and the file created is 16.3 mb in size.

   The output of similarity matrix is stored under the directory /bigd27/partAsimilarity_MatrixFinal, whose execution took around 1.6 minutes and the file created is 6.7 mb in size.

Problem Description – 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 result as an Avro file

Solution Strategy – Part B

Avro is a data serialization system, which provides

- Rich data structures.
- A compact, fast, binary data format.
- A container file, to store persistent data.
- Remote procedure call (RPC)

1. The top 1000 words are taken from "/bigd27/thousandwordsniv" list and inverted index is created, program is named as "BAvroWrite.py ", which can be executed using the command

   Spark-submit –master yarn –num-executors 5 BAvroWrite.py

2. The inverted index is written in avro format, in the path "/bigd27/PartBInverted_IndexFinal.avro"

   The individual avro files will be stored in .avro format as shown below

   /bigd27/PartBInverted_IndexFinal.avro/part-00000-cdc4aa4f-4e73-4748-b76a-86d48cfceb5c-c000.avro

3. time taken to execute this program is 24 minutes and file size is reduced to 5.2 mb, which is significantly lesser than file size created in part A. so we can state that writing data in avro format saves the memory and execution is also faster.

4. In The second part of the Part b, the avro file will be read by BAvroRead.py program, can be executed using the command

   Spark-submit –master yarn –num-executors 5 BAvroRead.py

5. which will read the avro files and calculate the similarity matrix and writes the result in avro format again.

6. The similarity matrix can be found in the path

   "/bigd27/PartBSimilarity_MatrixFinal.avro"

   the execution time was very less than 1.6 minutes and the file size is 2.3 mb which is way lesser than the similarity matrix file created in first part.

Problem Description – Part C:

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

Solution Strategy – Part C

1. Parquet is a columnar storage format available to any project in Hadoop ecosystem, regardless of choice of data processing framework, data model or programming language ,Apache Parquet is built to support very efficient compression and encoding schemes
2. In this part the initial 1000 words are taken from "/bigd27/thousandwordsniv" list and inverted index is created
3. program is named as "CParquetWrite.py ", which can be executed using the command Spark-submit –master yarn –num-executors 5 CParquetWrite.py

4. The inverted index is written in Parquet format, in the path "/bigd27/PartCInverted_IndexFinal.parquet"

5. The individual parquet files will be stored in parquet format as shown below

   /bigd27/PartCInverted_IndexFinal.parquet/part-00000-cdc4aa4f-4e73-4748-b76a-86d48cfceb5c-c000.parquet

6. time taken to execute this program is 24 minutes and file size is reduced to 2.7 mb, which is significantly lesser than file size created in part A. so we can state that writing data in parquet format saves the memory and execution is also faster.
7. In The second part of the Part b, the inverted index parquet file will be read by CParquetRead.py program, can be executed using the command

   Spark-submit –master yarn –num-executors 5 CParquetRead.py

8. which will read the parquet inverted index file and calculate the similarity matrix and writes the result in parquet format again. The similarity matrix can be found in the path "/bigd27/PartCSimilarity_MatrixFinal.parquet"
9. The execution time was very less than 1.6 minutes and the file size is 4.5 mb which is way lesser than the similarity matrix file created in first part.

Problem Description – Part D:

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

Solution Strategy – Part D

In this part I am choosing to implement a version of part C that creates snappy compressed parquet files.

1. Parquet files are by default snappy compressed.
   Hadoop-snappy format is one of the compression formats used in Hadoop. It uses its own framing format as follows:
   - A compressed file consists of one or more blocks.
   - A block consists of uncompressed length (big endian 4 byte integer) and one or more sub blocks.
   - A sub block consists of compressed length (big endian 4 byte integer) and raw compressed data.

2. The initial 1000 words are taken from "/bigd27/thousandwordsniv" list and inverted index is created, program is named as "DSnappyParquetInv.py ", which can be executed using the command

   Spark-submit –master yarn –num-executors 5 DSnappyParquetInv.py

3. In the previous stage when the parquet file was created the compression argument is specified as none, so in the part C the data is not snappy compressed, while the part D the compression argument is not specified so it is compressed in snappy format

   As we can see from the below file format, data is stored in snappy format

   /bigd27/PartDInverted_IndexFinal.parquet/part-00003-f703e7d5-e057-45e3-8074-adf277554c5b-c000.snappy.parquet

4. The snappy format is efficient as it takes less execution time and it reduces the file size significantly

5. The second part of snappy compressed file involves reading data from the inverted index created in the previous step, the program DsnappyParquetSim.py has the code to achieve this

   Spark-submit –master yarn –num-executors 5 DSnappyParquetSim.py

6. The similarity matrix is calculated, and results are saved again in snappy compressed format

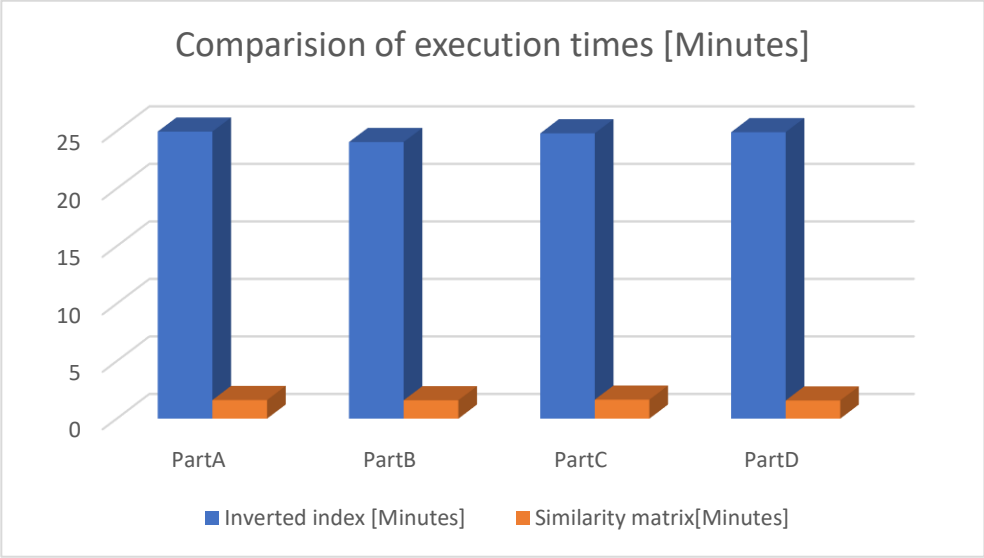
Problem Description – Part E:

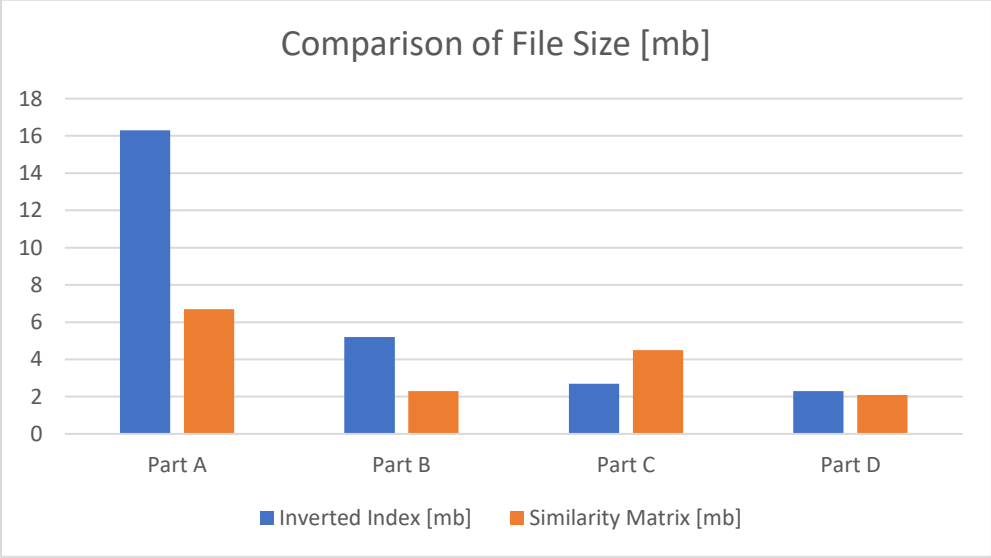
Compare file sizes for Parts a – d, as well as execution time of all versions of Part a-d using the corresponding format using the MEDIUM data set using 5 executors

Solution Strategy – Part E

Comparison Table

Sl no	Part A		Part B		Part C		Part D	
	Inverted index	Similarity matrix	Inverted index [Avro]	Similarity matrix [Avro]	Inverted index [Parquet]	Similarity matrix [Parquet]	Inverted index parquet snappy compressed	Similarity matrix Parquet Snappy compressed
Execution time	1498.8s	97.67s	1443.9s	96.04	1489.9	99.4s	1495s	95.12s
File size	16.3mb	6.7mb	5.2 mb	2.3 mb	2.7 mb	4.5 mb	2.3 mb	2.1mb





Comments on Observation:

As we can see from the above table the execution time and file size differ for different kind of compression technique

- The average execution time of Avro File is the lowest when compared to the other input files.
- The execution time of all the parts were similar while creating the inverted index.
- When we talk about the file size -There is significant reduction in size of the file created when we use parquet snappy compressed file format than all other formats, so snappy compressed method is best compression method which will reduce memory usage and increases the speed of execution
- The file size created using normal parquet file was more compared to parquet file snappy compressed format.

Measurement of size of files

Once the data files are written in the /bigd27/*** directory, there are two ways in which we can calculate the size of file created

First method:

hadoop fs -du -s -h /bigd27/Inverted_IndexMediumdatasetFinal

this command will specify the size of data file in mb

second method:

using -getmerge option , we have to merge all the different files into a text file and then determine the size of text file using -ls -l command , as shown below,

hdfs dfs -getmerge /bigd27/Inverted_IndexMediumdatasetFinal/part-00000 /bigd27/Inverted_IndexMediumdatasetFinal/part-00001 /bigd27/Inverted_IndexMediumdatasetFinal/part-00002/bigd27/Inverted_IndexMediumdatasetFinal/part-00003 /home2/cosc6339/bigd27/hw3/filesize.txt

Results:

Resources Used:

1. Whale Cluster:

- The clusters consist of a login node (whale.cs.uh.edu) and several compute nodes, The cluster shares home directories, but are otherwise separate.
- The access method to whale from the outside world is by using ssh using the command ssh -l -bigd27 whale.cs.uh.edu
- The login nodes are to be used for editing, compiling and submitting jobs.
- Program runs are submitted through Hadoop (framework that supports the distributed execution of large scale data processing) on this cluster.

- Jobs are submitted from the login node and run on 1 or more compute nodes. Jobs then run until they terminate in some way, e.g. normal completion, timeout, abort.

2. PySpark:

- The Spark Python API (PySpark) exposes the Spark programming model to Python.
- Key Differences in the Python API: There are a few key differences between the Python and Scala APIs:
    - Python is dynamically typed, so RDDs can hold objects of multiple types.
    - PySpark does not yet support a few API calls, such as lookup & non-text input files.
    - PySpark can also be used from standalone Python scripts by creating a SparkContext in your script and running the script using bin/pyspark
    - Functions can access objects in enclosing scopes

3.Python

- Python is an easy to learn, powerful programming language.
- It has efficient high-level data structures and a simple but effective approach to object oriented programming
- python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.
- The Python interpreter is easily extended with new functions and data types implemented in C or C++ (or other languages callable from C)
- Python is also suitable as an extension language for customizable applications

Description of measurements performed

the execution was carried out 2 times for each part A-D, each time there was a slight change in the values of execution time.

Comparison Table

Sl no	Part A		Part B		Part C		Part D	
	Inverted index	Similarity matrix	Inverted index [Avro]	Similarity matrix [Avro]	Inverted index [Parquet]	Similarity matrix [Parquet]	Inverted index snappy compressed	Similarity matrix Snappy compressed
Execution time	1498.8s	97.67s	1443.9s	96.04	1489.9s	99.4 s	1495s	95.12s
Execution time	1499.1s	98.6s	1443.8s	96.03s	1489.8s	101 s	1494s	95.09s
File size	16.3mb	6.7mb	5.2 mb	2.3 mb	2.7 mb	4.5 mb	2.3 mb	2.1mb
File size	16.3mb	6.7mb	5.2mb	2.3 mb	2.7mb	4.5 mb	2.3mb	2.1mb