BIG DATA THIRD ASSIGNMENT REPORT

SAI SRI VARSHA GADDE -- 1885164

Problem statement:

Given dataset containing measurements of pollutants from a large number of sensors for the year 2013 in the state of Texas. Each Sensor reports the measurements every 5 minutes. Dataset has data regarding all the pollutants but we concentrate only on O3(Ozone) pollutant which can cause damage to the people.

There are two input files for this assignment, one is 2013_data_jan.csv file, the data collected from month of January 2013 which is roughly 780MB and other file is 2013_data_full which is roughly 9 gigabytes of data.

As a part of this assignment, need to develop a Pyspark code to calculate hourly average of O3(ozone) concentration only for each site separately. As this data is an actual data from sensors, all entities cannot be used for processing. So, consider only data points that have an empty flag and data is not useful if the flag value is set and also eliminate null values and garbage values (negative values) as well.

Convert csv file format to other file formats (Parquet, HDF5, JSON, Avro) and compare the file size of the new formats and calculate the hourly average with these new file formats as input and compare the execution times for large data file.

Solution Strategy:

Apache Spark is a lightning-fast cluster computing technology, designed for fast computation. It is based on Hadoop Map Reduce and it extends the Map Reduce model to efficiently use it for more types of computations, which includes interactive queries and stream processing.

Spark provides built-in APIs in Java, Scala, or Python. We can write applications in different languages. For this assignment I have used PySpark in order to implement Apache spark.

Part1: Calculation of Hourly Average:

Step1: Imported the modules, Spark Session from pyspark which i have used in order to parallelize the task and I have first read the csv file into a pyspark dataframe. PySpark DataFrames which are tables that consist of rows and columns of data. It has a two-dimensional structure wherein every column consists of values of a particular variable while each row consists of a single set of values from each column.

Step2: After reading the data into dataframe, I have renamed the column names using withColumnRenamed(), which is used to rename a column or multiple columns in a dataframe. I have also changed the datatype of value from String to double as average will be calculated on numeric values not on strings.

Step3: Next I have applied “filter” to dataframe for extracting only O3 concentration and also for considering only flag values which are null.

df1.filter((df1._c7=="o3") & (df1._c11.isNull()))

As hourly average has to be calculated only for O3 concentration, which is harmful and also only empty flag need to be considered as flag indicates problems.

Step4: Identify all null values which are improper to calculate hourly average and replace them with -1 and I have replaced all the null values in column _c10 with a constant - dftest.withColumn("_c10", when(col("_c10").isNull(),-1).otherwise(col("_c10")))

Step5: All the negative values are also considered garbage values, So I have filter out all the positive values from the _c10 column, making sure data is clean and there are no garbage values.

dftest2.filter(dftest2._c10>0).

Step6: Next I have applied groupby("Year", "Month", "Day", "Hour","site"), as hourly average has to be calculated for each site for each hour for each month in an year and then calculated the average using the aggregate function “avg”. (avg(“_c10”)). In this way I have calculated the hourly average for each site accordingly.

Next I have applied orderby() function so that output is in ordered format and it is displayed ordered by site, month, day, hour.

Step7: After, I have redirected the output to an output directory using df.write.option() and format of the output files is CSV.

Next I have executed this pyspark code(averagehour.py) to calculate hourly average for each site using 1,2,4, and 8 executors using 2 cores per executor for both 2013_data_jan.csv and 2013_data_full.csv

Part2: Pyspark code to convert CSV file to Parquet:

Apache Parquet is designed for efficient as well as performant flat columnar storage format of data compared to row based files like CSV. Parquet uses the record shredding and assembly algorithm.

I have chosen this approach because parquet is good for queries that need to read certain columns from a large table. Other Reason is Parquet can only read the needed columns therefore greatly minimizing the IO.

Step1: I have created spark session object and using this object I have read the 2013_data_full csv file and stored it in a data frame.

Step2: Next I have used df.write.option ("overwrite"), I have used mode overwrite so that file will be overwritten whenever pyspark code is executed.

I have used repartition(1) method in the dataframe to redirect the whole output to a single file.

Depending upon change the number in repartition() method, the output directory will have those many parts of the parquet file. I have used only a single partition as I want the output as single parquet file.

df.repartition(1).write.option ("overwrite").parquet("output directory") --- Next redirected the parquet file to output directory.

Pyspark code to convert CSV file to Json:

JSON is language independent and light weight text based interchange format. Spark SQL can infer the schema of a JSON dataset and load it as a DataFrame.

I have created spark session object from pyspark.sql and using this object I have read the 2013_data_full csv file and stored it in a data frame.

I have used repartition(1) method in the dataframe to redirect the whole output to a single file.

Depending upon change the number in repartition() method, the output directory will have those many parts of the json file. I have write.json() method to convert to JSON format and redirected to output directoty.

df.repartiton(1).write.json("output directory");

Part3: Parquet file as input to calculate the Hourly average:

I have read the parquet file from the output directory using spark object and read methods.

Spark.read.parquet("/home/output/stud05/parquetjan/parquetfile.parquet"). I have read this parquet file into a data frame.

Next I have performed the same computations like csv file to calculate hourlyaverage for each site, first filtered out O3 concentration and flag Not null values and handled the garbage values by -1 and filtered them out and then applied groupby() operation and calculated the average aggregate function. To order the output I have used orderby() function.

Next I have written the output parquet files to a directory.

df.write.parquet("output directory")

Next I have executed this pyspark code(parquetaveragehour.py) to calculate hourly average for each site using 1,2,4, and 8 executors using 2 cores per executor for 2013_data_full.csv.

JSON file as input to calculate the Hourly average:

I have read the JSON file from the output directory using spark object and read methods.

Spark.read.json(“/home/output/stud05/data_full.json/parquetfile.json”). I have read this JSON file into a data frame.

Next I have performed the same computations like csv file to calculate hourlyaverage for each site, first filtered out O3 concentration and flag Not null values and handled the garbage values by -1 and filtered them out and then applied groupby operation and calculated the average aggregate function. To order the output I have used orderby function.

Next I have written the output JSON files to a directory.

df.write.json(“output directory”)

Next I have executed this pyspark code(jsonaveragehour.py) to calculate hourly average for each site using 1,2,4, and 8 executors using 2 cores per executor for 2013_data_full.csv.

Description of how to run your code:

Part1: To calculate hourly average for both CSV files 2013_data_jan.csv(780mb) and 2013_data_full.csv(9Gb)

My pyspark code file name is **averagehour.py**. I have used this file for both data sets Jan and Full CSV files. In averagehour.py file I have written both input directory to read the file from and output directory to write the file to.

For **2013_data_jan.csv** file: Spark.read.csv(“/home2/input/2013_data_jan.csv”) and df.write.option(“header”,”true”).csv(“/home/output/stud05/janoutput4”)

For **2013_data_full.csv** file: Spark.read.csv(“/home2/input/2013_data_full.csv”) and df.write.option(“header”,”true”).csv(“/home/output/stud05/fulloutput4”)

Varying the number of executors:

stud05@crill>spark-submit --master spark://crill:28959 --total-executor-cores 2 --executor-cores 2 ./averagehour.py (Number of executors 1 , so total—executor-cores 2)

executor-cores per executor remains same i.e 2 only number of total-executor-cores get changed based on number of executors.

Total-executor-cores= 2(executor-cores) * 1((number of executors)=2

And specify the pysparkcode file name that you want to execute.

stud05@crill>spark-submit --master spark://crill:28959 --total-executor-cores 4 --executor-cores 2 ./averagehour.py (Number of executors 2 , so total—executor-cores 4)

stud05@crill>spark-submit --master spark://crill:28959 --total-executor-cores 8 executor-cores 2 ./averagehour.py (Number of executors 4 , so total—executor-cores 8)

stud05@crill>spark-submit --master spark://crill:28959 --total-executor-cores 16 executor-cores 2 ./averagehour.py (Number of executors 8 , so total—executor-cores 16)

Part2: Same Spark submit command is used for parquet and Json conversion, we can specify the number of executors as per wish. I have used 4 executors while converting CSV file to JSON and Parquet for 2013_data_full.csv

My pyspark code files for csv to convert to parquet, JSON converison --- **parquetconverison**.py, **jsonconverison**.py

Part3: Parquet file to calculate hourlyaverage is – **parquetaveragehour**.py. Inside this file I have written code to read the converted parquet file from output directory and write the output to a directory.

JSON file to calculate hourlyaverage is – **jsonaveragehour**.py. Inside this file I have written code to read the converted JSON file from output directory and write the output to a directory.

Same spark submit command is used by varying number of executors 1,2,4 and 8 for parquet and JSON pyspark code for 2013_data_full file.

All my source files are in /home2/stud05.

Resources Used:

We have used crill cluster(crill.cs.uh.edu) to execute all our programs:

Cluster information:

- cluster has 1 front end node and 24 compute nodes
- 8 compute nodes have been reserved for our assignment
    - 2 AMD nodes with 24 cores and 32 GB memory each
    - 3AMD nodes with 8 cores and 32GB memory each
    - 3 Intel nodes with 16 cores and 24GB memory each

Software used for code development:

- Python 2.7
- Pyspark verison 2.3.4

Measurements Performed and Results:

- **Part1**: Measured the execution times for pyspark code(averagehour.py) to calculate hourly average for each site using 1,2,4, and 8 executors using 2 cores per executor for both CSV files as input 2013_data_jan.csv and 2013_data_full.csv.

Execution Times for calculating hourly average for **2013_data_jan.csv** file with 1,2,4,8 executors

Number of Executors	**Executor cores**	**Total executor cores**	**Execution times (seconds)**
1	2	2	215.979
2	2	4	163.5844
4	2	8	148.5650
8	2	16	135.409

Execution Times for calculating hourly average for **2013_data_full.csv** file with 1,2,4,8 executors

Number of Executors	**Executor cores**	**Total executor cores**	**Execution times(seconds)**
1	2	2	1535.3816
2	2	4	519.2827
4	2	8	392.6531
8	2	16	245.7052

<u>Findings:</u>

For Jan File and full CSV files,I observed that there is a variance in execution times depending upon the number of executors.

For Jan CSV file – Execution time for 1 executor took 215.979 seconds and it got reduced for 2 and 3, where 2 executors took 163.58 and 4 executors took 148.565 and it got more decreased for 8 executors to 135.409.

For Full CSV file – Execution time for 1 executor took 1535.3816 seconds and it got drastically reduced for 2 it took 519.2827 and 4 executors took 392.6531 and it got again decreased for 8 executors to 245.7052.

I observed that, As Number of executors increases, keeping the executor cores constant 2 per executor, Execution times gets decreased. Number of executors and Execution times are to be inversely proportional to each other.

Comments: As the number of executors increases in pyspark dataframe total number of executer cores also increases which increases the number of partitions, these increased partitions will increase parallelism level. Having less partitions is not beneficial as some of the worker nodes will be idle resulting in less concurrency and hence more execution time. So, when we increase the number of executors, partitions will get increases and workload will be shared among the worker nodes, as the work is more distributed, tasks will be finished with quick time and hence overall resulting in lesser execution time.

So as number of executors increases, amount of parallelism also increases with more number of partitions which results in lesser execution time.

Part 2:

- Compared the file size for after converting CSV file(2013_data_full.csv) to parquet file format. Code to convert to Parquet file is in parquetconversion.py
- Compared the file size for after converting CSV file(2013_data_full.csv) to JSON file format. Code to convert to Parquet file is in jsonconverison.py

File Formats	File Size(2013_data_full)
CSV	9 GB
Parquet	525 MB
JSON	21 GB

<u>**Findings:**</u>

Parquet: I observe that when compared to CSV file size, parquet file size is drastically reduced. Parquet is column-oriented structure and it has efficient columnar storage like blocks, row groups of data (horizontal partitioning of data in to rows) , column chunks of data, pages rather than only row based data like CSV. If we want to read only certain columns of data not the entire row, parquet is efficient to read only a part of record. When querying, columnar storage you can skip over the non-relevant data very quickly.

Parquet is a self-describing structured data format that embeds the schema within the data itself and also its column oriented data structure makes the parquet file optimized and compressed (less size) and also minimizes Input/output.

JSON: I observe that when compared to CSV file size, JSON file size is drastically increased. JSON is also row oriented data format but data is stored as unordered set of name/value pairs separated by comma and each object contains field names and has special characters for syntax. Because of all these extra fields and characters, it adds an extra space and there is no schema related structure, all these increase the size of the file. JSON also contains nested structures and javascript datatypes and this script requires external metadata to identify datatypes, all these factors summing will increase the size of JSON file.

Part 3:

- Measured the execution time for full data set (2013_data_full) for parquet converted file format and JSON converted file format with 1,2,4, 8 executors
- Compared the execution times for full data set for CSV, Parquet and JSON file formats with different executors.

Execution Times for calculating hourly average for **2013_data_full** dataset with 1,2,4,8 executors for parquet file (parquetaveragehour.py)

Number of Executors	Executor cores	Total executor cores	Execution times(seconds)
1	2	2	154.7934
2	2	4	115.0508
4	2	8	107.2596
8	2	16	102.2596

Execution Times for calculating hourly average for **2013_data_full** dataset with 1,2,4,8 executors for JSON file format (jsonaveragehour.py)

Number of Executors	Executor cores	Total executor cores	Execution times(seconds)
1	2	2	1788.2736
2	2	4	1387.7292
4	2	8	1019.3924
8	2	16	791.5423

Findings:

For full dataset, I observed that for both parquet file and JSON file there is a variance in execution times depending upon the number of executors.

For parquet file, Execution time for 1 executor took 154.7934 seconds and it got reduced for 2 and 3, where 2 executors took 115.0508 and 4 executors took 107.2596 and slightly reduced again for 4 executors to 102.2596. Slight decrease when as the number of executors gets increased.

For JSON file, Execution time for 1 executor took 1788.2736 seconds and it got reduced for 2 to 1387.7292 and slightly decrement for 3 executors took 1019.3924 and for 4 executors it got reduced again to 791.5423.

I observed that, As Number of executors increases, keeping the executor cores constant 2 per executor, Execution times gets decreased.

Comments: Total number of executor cores increases as number of executors increases in Pyspark dataframe which in turn increases the number of partitions, these increased partitions will increase parallelism level workload will be shared among the worker nodes and time will be reduced to complete the task. Having less partitions is not beneficial as some of the worker nodes will be idle resulting in less concurrency and hence more execution time.

So number of executors increases, amount of parallelism also increases among the worker nodes and this results in lesser execution time.

Comparison of execution times for full data set using File formats CSV, Parquet, JSON with 1,2,4,8 executors

Number of Executors	Executor cores	Total executor cores	Execution Times(seconds)		
			CSV	Parquet	JSON
1	2	2	1535.3816	154.7934	1788.2736
2	2	4	519.2827	115.0508	1387.7292
4	2	8	392.6531	107.2596	1019.3924
8	2	16	245.7052	102.2596	791.5423

Findings:

I observed that parquet has less execution times when compared with all the three file formats (CSV, Parquet and JSON). JSON has higher execution times and CSV execution times are in between the parquet and JSON execution times for 1,2,4 and 8 executors.

Comments: When csv file is converted to parquet file, size of the file will be drastically reduced and if the file size is reduced then the time taken to process the file will also be reduced and similarly for JSON file as the file size is increased incrementally, execution time for processing the file will also increase.

- CSV is human-readable and easy to edit manually provides simple schema and csv is good to work with flat data rather than complex data structures.
- JSON is comparatively better than CSV while working with the large volume of data and in terms of scalability of files or application where CSV is good enough in working with small files and fewer data. CSV is mainly used only when there is a requirement of sending the large volume of data and there is an issue with bandwidth. CSV cannot be used where the data is complex and unstructured, then only JSON is the better option to

work with complex data. In JSON, it is significantly easier to work within and used for the programming languages but JSON file becomes twice the CSV file when a lot of data is required or maintained in files, use JSON when storage is not a concern.

- Unlike CSV and JSON, parquet files are binary files that contain metadata/schema related structure about their contents. Therefore, without reading/parsing the contents of the file(s), Spark can simply rely on metadata to determine column names, compression/encoding, data types. Column metadata for a Parquet file is stored at the end of the file, which allows for fast, single-pass writing.
- Parquet is good choice for heavy workloads when reading portions of data. Parquet is well suited for data storage solutions where aggregation on a particular column over a huge set of data is required.

To conclude, Parquet has smaller file size and best execution times as parquet is column-oriented structure self-describing structured data format that embeds the schema within the data itself and When querying, columnar storage you can skip over the non-relevant data very quickly and gives lesser execution times. JSON has bigger file size and higher execution times as JSON as unordered set of name/value pairs separated by comma and requires external metadata for datatypes and also syntax representation makes it bigger size and it takes time to parse such huge file.CSV file in between execution times/moderate and it works better with flat data rather than complex data structures.