

Project 1: File Formats

Mahsa Rezaei Firuzkuhi

mrezaeif@cougarnet.uh.edu

Problem Description

In this problem we have a large sensor data set in the comma separated value (CSV) format in which each sensor is reporting a measurement every five minutes for various concentration parameters of the pollutants and we need to identify the hourly average of only one of the concentrations (O3) based on the input data set that we have. This problem is challenging because the dataset consists of actual data collected from the sensors and there is many data points or entries that cannot be used for information processing and on the other hand there are various data point values that are missing (the sensor data was not collected every 5 minutes). Every single data has a flag indicating the availability of the data point value. Many of the flag values can be set indicating that the sensor could not provide the actual/correct data point value at that point in time due to having the sensor calibration cycles or etc. Finding the actual hourly average of each concentration on each site is a highly challenging task because we are interested only in O3 concentration hourly average value information, and we are not interested in any other pollutant or any other measurement. We need to identify which values we can use to calculate the hourly average and then performing the actual calculation. To do it in Pyspark code we need to decide which intermediate keys to be chosen to combine all the measurements of the data. If we need to calculate the hourly average of a specific concentration for every hour of a year, the key should contain month, day and hour values corresponding and the measurement after ensures that the reducer will get all the data points for each of these intermediary keys. There are situations where we do not have a single valid entry within a single hour, in those cases we provide a constant value (-1). In part 2 we are utilizing other file formats to find the optimal file format for our application. The CSV file format is suboptimal in many aspects. So, another goal is to compare different file formats in terms of timing and file size. We consider two different file formats from the original CSV format: the JSON and Parquet file formats and compare the size of their related files and the time our code (O3 hourly average calculation) requires reading data from these formats, while

having 1, 2, 4 and 8 number of executors in part 2 and 3 of the assignment. So, we used Spark technology to be able to do the operation in parallel form. In the following section we briefly introduce the spark technology and data-frame and how we employ its features to speed up the hourly average calculation of O3 concentration and a brief explanation of these two file formats.

Solution Strategy

Apache Spark is an open-source cluster-computing framework. Spark provides an interface for programming entire clusters with implicit data parallelism and fault-tolerance. Apache Spark provides programmers with an application programming interface centered on a data structure called the resilient distributed dataset (RDD), a read only multi-set of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way. Spark uses shared memory to facilitate communication between resources and this fact makes it faster compared to other technologies that do not use shared memory like Hadoop. Data Frames are distributed collection of rows under named columns and conceptually like a table in a relational database.

We process two datasets of sensor-measured data points: the first one contains data only from the month of January in year 2013 and the second full database of the year 2013. Every single data point of the data set contains 12 columns: year, month, day, hour and minute of when the measurement was performed, and measurements has been taken every five minutes by sensors. Some of the measurement parameters like “region” are not useful in calculating the hourly average of the data in each site. Then we have the parameter name and identification which we are interested in to identify the name “o3” only. The next two columns are the location of the sensor (site) and sensor ID which we are interested in because a single site can have potential multiple sensors. The last two value are the most important columns including the actual sensor measurement value (which we are going to evaluate the average of that in every hour) and the flag which indicates whether that sensor value is valid or not. If the flag is empty (null) it indicates the value is valid otherwise that data point should be discarded and not considered in calculation of the average. The solution for solving this problem is described in the following steps. We need to implement a Pyspark code solution which is able to calculate the hourly average of O3 value for each “site”. In the first step we read all the single data points for all pollutants in one year and in

different sites in a data-frame structure by following command and then filter the data frame with O3 concentrations:

```
schemaString = "Year Month Day hour min region parID parName site cams value flag"
```

```
df=sqlContext.read.format('com.databricks.spark.csv').options(delimiter=',',nullValue=").load("/home2/input/2013_data_full.csv",schema=schema)
```

```
df = df.filter(df.parName == "o3")
```

There are several conditions which we must consider before the calculation. The first one is the flag to be set which means the value should be discarded and not to be considered. The other scenario which we must consider is where the flag is not set (flag it is empty) but the value is garbage (negative value which is a wrong measurement in our case). In those cases, we must discard the value and not using that for calculating the average. In other word, we consider this scenario equivalent to the scenario in which the flag is set for that data. Since the sensor take the measurements every 5 minutes so in an hour, they will produce at least 12 values (in some cases the whole data point is duplicated). So we need to take average of those 12 or more values using the `groupBy()` function and replace those 12 rows with the averaged row in the final data frame and the final data frame will have only 5 columns.

```
Df_groupedAvg = df.groupBy("site", "Month", "Day", "hour").agg(F.mean('value')).alias('Avg_value')
```

There will be several boundry conditions which we have to consider before performing the `groupBy()`:

- 1- If all the 12 values for a specific hour are negative values and their flags are empty (not set), then for that hour the average should be “-1”
- 2- If for an specific hour, there is even a single positive value, then that hour should have an average in the final data frame
- 3- If for all the 12 values for a specific hour the flag is set (flag is not empty), then for that hour the average should be “-1”

We have created different data-frames for different conditions and then we did `Union ()` of those conditions at the end to get the final data-frame.

In part 2 we are utilizing other file formats to find the optimal file format for our application. The CSV file format is suboptimal in many aspects, and we are going to select two additional file format and provide the pyspark code that converts the CSV file format to those two other formats. We have selected the JSON and Parquet and using the following functions to convert the csv full input dataset to Parquet and JSON file formats. The advantage of the binary file format over the text file formats is that the binary files is much smaller than the text file when we have numerical values. The downside of the binary file format is that the binary file is not human readable. The Parquet and HDML file formats keep data in binary format internally but also store some additional information to maintain some structure and schema of the data file. Parquet has a columnar format. Although it reduces a lot of I/O cost to make great read performance, it is computationally intensive on the “write” operation. It can add new columns to the end of the structure but does not fully support schema evolution. It supports compression and efficient encoding schemes. On the other hand, serialization formats which are based on the text file format like XML and JSON store the schema of the data structure along with the actual data itself. The main downside of these two files format is that they are text-based file formats, and the schema is stored not only one time but every single time that a particular data item is stored in the file.

Parquet:

```
df.coalesce(1).write.parquet('/home/output/stud25/project-1_parquetRecords', 'overwrite')
```

```
df = sqlContext.read.parquet('/home/output/stud25/ project-1_parquetRecords.parquet')
```

JSON:

```
df.coalesce(1).write.json('/home/output/stud25/ project-1_jsonRecords', 'overwrite')
```

```
df = sqlContext.read.json('/home/output/stud25/ project-1_jsonRecords.json')
```

Description of how to run your code

To run the code, named “project1.py”, please use the following commands. To change the input and output dataset directories you need to change those values inside the source code. All other parameters can be set using the following command. For example, we must change the number of executors, it means changing the value of total-executor-cores in the spark-submit command (we consider the value 2 for the number of executors per core in all the measurements):

```
spark-submit --master spark://crill:28959 --total-executor-cores 8 --executor-cores 2 ./project1.py
```

For 1 executor we insert **--total-executor-cores 2**

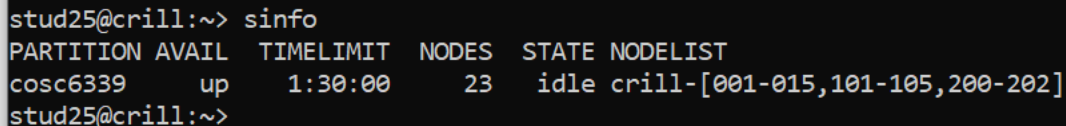
For 2 executor we insert **--total-executor-cores 4**

For 4 executor we insert **--total-executor-cores 8**

Results

Description of resources used

We have utilized 23 compute nodes of the “Crill” cluster for our project. The properties of these nodes are listed below:



```
stud25@crill:~> sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
cosc6339   up       1:30:00    23   idle crill-[001-015,101-105,200-202]
stud25@crill:~>
```

2 Appro 1326G4 nodes (crill-101 - crill-102), each node with

- two 2.2 GHz 12-core AMD Opteron processor (24 cores total)
- 32 GB main memory
- four NVIDIA Tesla M2050 GPUs (448 cores each) (not operational as of today)
- 4xQDR InfiniBand HCA

3 Sun Microsystems X2200 nodes (crill-103 - crill-105), each node with

- two 2.2 GHz 4-core AMD Opteron processor (8 cores total)
- 32 GB main memory
- 4xQDR InfiniBand HCA

3 HP DL 160 Gen 8 nodes (crill-200 - crill-202), each node with

- two 2.4 GHz quad-core Intel Xeon E5-2665 processor (12 cores/24 threads total)
- 16 GB main memory
- 4xQDR InfiniBand HCA

Network Interconnect

- 288 port 4xInfiniBand QDR InfiniPath Switch (donation by TOTAL)
- 24 port 4xInfiniBand SDR switch (I/O switch to the SSD storage)
- 48 port Netgear GE switch

Storage

- 4 TB NSF /home directory
- 18 TB distributed storage operated as OrangeFS (/pvfs2) or BeeGFS (/beegfs)
- 2 TB RamSan 620 SSD storage (/pvfs2-ssd) (not operational)

Description of measurements performed

We imported the time library and used `time.time()` to clock the execution time. In the following three sections we will describe the details of the measurements. We present the average of the three different running of the code in the following tables. The sections one, two and three of the assignment is about doing the performance evaluation and changing the various parameters of the code.

Part 1:

We need to change the number of executors that we are using for measurements. We consider both small and large database files and running the Spark code with one, two, four and eight number of executors and evaluate how does the performance will change. We consider two cores per executors and keep this value fix in all the measurements. We expect that the more Spark executors we are using, the smaller the execution time becomes. This performance improvement is dependent on the extra hardware employment for execution. We have measured the execution time and provide the results in the following table for different number of executors And for each number of executors, we ran the code several times (at least 3 execution) and provide the average execution time.

Number of Executors	1	2	4	8
Execution Time (Full-DB)	1062	479	249	137
Execution Time(Jan-DB)	87	55	41	30

Table 1: Average execution time for different number of executors for both Full year and January datasets

Finding:

Result shows that, increasing the number of executors is decreasing the execution time. We need to consider that increasing number of executors reduces amount of time spent on each node but, it will increase shuffling overhead so increasing number of executors is not helpful in all the cases but in our case the execution time has significantly decrease. In our example, the amount of input data was big enough to utilize extra computing power available with higher number of executors. One important way to increase parallelism of spark processing is to increase the number of executors on the cluster. However, knowing how the data should be distributed, so that the cluster can process data efficiently is extremely important. As we can observe from the execution time while varying the number of executors from 1 to 2 to 4 and 8, the execution time has been decreased significantly for both the Full database and Jan database. While observing the execution time for larger databases, shows that increasing number of executors was much more effective in this application with larger datasets in comparison to small database.

Part 2:

In part 2 we are utilizing other file formats to find the optimal file format for our application with full dataset. The CSV file format is suboptimal in many aspects, and we are going to select two additional file format and provide the pyspark code that converts the CSV file format to those two other formats. We have selected the JSON and Parquet.

File Format	CSV	Parquet	JSON
File Size (CSV)	541MB	517MB	24GB

```
drwxrwxrwx  2 stud25  cosc6339   4096 Nov 13 12:51 hw3_2_avroRecords
drwxrwxrwx  2 stud25  cosc6339   4096 Nov 13 15:17 hw3_2_jsonRecords
drwxrwxrwx  2 stud25  cosc6339   4096 Nov 13 14:10 hw3_2_parquetRecords
(sftp> cd hw3_2_jsonRecords/
(sftp> ls -l
-rw-r--r--  1 stud25  cosc6339      0 Nov 13 15:17 _SUCCESS
-rw-r--r--  1 spark   wheel  24062554538 Nov 13 15:17 part-00000-0b7a754d-2abb-42af-a7be-57c9864555b2-c000.json
(sftp> cd ..
(sftp> cd hw3_2_parquetRecords/
(sftp> ls -l
-rw-r--r--  1 stud25  cosc6339      0 Nov 13 14:10 _SUCCESS
-rw-r--r--  1 spark   wheel  542804354 Nov 13 14:10 part-00000-b80b8cf4-1e4a-4a01-8c5a-4e42ff787c79-c000.snappy.parquet
(sftp>
```

Table 2: Dataset file size for different file formats in Section 2.

Finding:

Based on table 2, CSV and Parquet with snappy compression have the minimum file size while the JSON is much larger. The advantage of the binary file format over the text file formats is that the binary files are much smaller than the text files when we have numerical values. The downside of the binary file format is that the binary file is not human readable. The Parquet and HDML file formats keep data in binary format internally but also store some additional information to maintain some structure and schema of the data file. Parquet has a columnar format. It reduces a lot of I/O cost to make great read performance. On the other hand, serialization formats which are based on the text file format like XML and JSON store the schema of the data structure along with the actual data itself. The main downside of these two file formats (XML and JSON) is that they are text-based file formats, and the schema is stored not only one time but every single time that a particular data item is stored in the file. As we can observe from the converted file sizes JSON size is much larger than the CSV and Parquet (Parquet with snappy compression have the minimum file size).

Part 3:

In this section, we change number of executors that we are using for measurements. We consider only large database(full) files and running the Spark code with one, two, four and eight number of executors and evaluate how does the performance will change. We consider two cores per executors and keep this value fix in all the measurements. We expect that the more Spark executors we are using, the smaller the execution time becomes. On the other hand, we are comparing the

execution time of the O3 hourly average calculation code while reading the dataset from different file formats that we have selected in part 2.

Number of Executors	1	2	4	8
Execution Time (CSV-DB)	1062	479	249	137
Execution Time(Parquet-DB)	589	380	50	80
Execution Time(JSON-DB)	1760	913	655	358

Finding:

Result shows that, as we expected to be like the trend in part 1 results, increasing the number of executors is decreasing the execution time. We need to consider that increasing number of executors reduces amount of time spent on each node but, it will increase shuffling overhead so increasing number of executors is not helpful in all the cases but in our case the execution time has significantly decrease. In our example, the amount of input data was big enough to utilize extra computing power available with higher number of executors in all three file formats of input dataset.

On the other hand, if we compare the execution time in case of CSV with Parquet and JSON file formats (for a fixed number of executors), obviously the execution time in case of using the parquet file format is the minimum and in case of JSON file format it is maximum and performance of the code in case of CSV file is in between. Parquet is a self-describing data format that embeds the schema or structure within the data itself and in our program, we are using this columnar characteristic, so we are paying back the overhead. The main downside of the JSON files format is that it is text-based file format, and the schema is stored not only one time but every single time that a particular data item is stored in the file and in our experiment, we did not utilized JSON columnar feature enough to gain performance.