# Big Data Analytics

# Assignment 3

**Arjun Subramanyam Varalakshmi (1546081)**

**Problem Description:**

In this assignment, we would be exploring different Hadoop supported file formats by comparing their space utilization and the performance of the spark in terms of execution speed while working on those different file formats. This assignment consists of two parts, which are described below:

In first part, we will be developing a Spark code using PySpark API which convert the CSV file into 3 different file formats i.e. i). Parquet file format ii). Sequence file format iii). JSON file format and compare the size of the generated files to the original input file.

In second part, we will be developing a separate PySpark code for four file formats i.e. CSV, JSON, Parquet, Sequence file formats to determine the percentage of delayed flights per Origin, and compare the performance of the code for different number of executors i.e. 5, 10 and 15 executors.

In this assignment, we would be using Apache Spark along with Hadoop Framework (HDFS, YARN) to complete the assignment.

**Solution Strategy:**

We would be solving the above problem using Apache Spark cluster computing framework and its advanced data structures such as RDD (Resilient Distributed Dataset) and DataFrames to solve the above problems.

The reason for choosing Apache Spark over other cluster computing frameworks is its ability to provide maximum utilization of resource without compromising on key features such as fault tolerance, scalability and high performance. It also uses facilitates python API using PySpark framework, which provides simple python interface for writing the code and provides various configuration options to optimize the execution.
There are different possible solutions which can be provided using the PySpark, the approach followed by me to solve this problem is as described below:

**<u>Step 1:</u>**

Input: HDFS directory containing flight data: /cosc6339_s17/flightdata-full

Output:

Directory containing files for each file format document:

/bigd21/full_data_csv/*
/bigd21/full_data_json/*
/bigd21/full_data_parquet/*
/bigd21/full_data_sequence/sequencefile/*

Key-Steps:

- Using the instance of SQLContext call read.csv method which reads all the csv files in the input directory and produces a Spark DataFrame containing the data along with its schema (by setting header=True).

- Perform transformations on the DataFrame to replace 'NA' values with None, so that can avoid error while casting the columns to their respective data types.

- Cast the columns to their data types (this would be helpful as this would avoid casting the data in the step 2).

- Save the DataFrame in different file formats.

Note: In the Part-1 even though we already have csv data available I'm saving the DataFrame in csv format also as this avoids removing NA values for the CSV format again in Step 2.

**<u>Step 2:</u> Four different versions of code are created for each file format; all 4 formats work in similar way as described below**

Input:

/cosc6339_s17/<directory containing file of one format>

Output:

Files containing percentage delay per origin

/bigd21/<directory containing text file with origin and delay percentage>

Key-Steps:

Using DataFrame.read.<file format> get the data from multiple files in the input directory onto a DataFrame.

In the DataFrame check if the value of column “DepDelay” (i.e. departure delay is greater than 0), and add a new column named “is Delayed”, if the value of DepDelay > 0 set value of isDelayed to 1 else 0.

Group the data per origin basis and sum the values of isDelayed and divide it by total number of entries per origin which would provide us with the delayed percentage value.

Note: the logic part would remain same for all the file format, the only thing that varies Is the way we read these files.

**Alternate Solution:** Read all the input files using WholeTextFiles and the convert the RDD to different file formats and perform the analysis.

Why I didn’t use alternate solution?

DataFrame provides better performance compared to RDD by following optimal transformation plan. Also, we can describe and analyze the data better with DataFrame compared to RDD.

**Code Explanation**:

**Step 1:**

- We would provide a directory path consisting of list of CSV files containing flight data as input to the program.
- sqlContext.read.csv method is used to read input and produces a DataFrame consisting of data and its schema.
- Get the column list from the DataFrames using DataFrame.columns.
- Check the value of each column for the presence of ‘NA’ values and replace them with None. By using withColumn method.
- Cast the columns to their correct data types as we didn’t infer schema while creating DataFrame due to presence of ‘NA’ values.
- Use DataFrame.write. <file format> method to write the data to different file formats.
- For sequencefile format use saveAsNewAPIHadoopFile method to convert the DataFrame to sequencefile format.

**Step 2:**

**Assumption:** Since the question didn’t mention whether to compute arrival delay or departure I am considering it to be departure delay as we are finding delay from origin and computed the delay.

- In the Second part, we would be compute the flight delay percentage per Origin.
- We would provide a directory path consisting flight data of input format to the version of code, E.g. Parquet file directory is provided for code written for parquet format.

- In the DataFrame check if the value of column “DepDelay” (i.e. departure delay is greater than 0), and add a new column named “is Delayed”, if the value of DepDelay > 0 set value of isDelayed to 1 else 0.
- Perform group by operation on the DataFrame on the “Origin” column and use agg operation and compute the sum of isDelayed column and divide the sum by count of entries per origin.
- Save the result as text file using DataFrame.saveAsTextFile

**Alternate Solution:** We could have considered arrival delay as delay interval also and computed the delay or we could have taken difference between scheduled time and scheduled departure time, if the difference > 0 set isDelayed column. As the time interval provided in DepDelay column was same as the difference I proceeded with using DepDelay value.

## **How to run the code**:

1. Unzip the HW3_1546081.zip folder and place the contents of the folder in the whale cluster.
2. cd to HW3 directory, it consists of a shell script named run_hw3.sh, to run the shell script give below command in terminal:
    a. ./run_hw3.sh
    b. If in case it gives permission error change permission using below command and execute it
        i. chmod +x run_hw3.sh (or chmod 777 run_hw3.sh) and execute.
3. The shell script would provide you with following 4 options:
    a. Run step1, step2, step3, step 4, step 5, run all and quit.
    b. If first 5 options are selected the script asks the user to enter the number of executors he would like to use to execute the code with.
4. If in case it’s required to change input directory path or how the input is given you need to edit the shell script and execute.

## **Results:**

## **Description of Resources used:**

The Assignment – 3 of this course is executed on Whale cluster, the components of Whale cluster can be as described below:

It consists of 57 Appro 1522H nodes

Hardware Details:

- CPU: Two 2.2 Ghz quad core AMD Opteron processor (8cores total)

- RAM: 16 GB
- Ethernet NIC: Gigabit Ethernet
- 4xDDR InfiniBand HCAs

Network interconnect:

- 144 port 4xinfiniBand DDR Voltaire Grid Director ISR 2012 switch (donation from TOTAL, shared with crill)
- two 48 port HP GE switch

Storage:

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

Software Details:

- Hadoop Framework version 2.7.2 (distributed – mode)
- Apache Spark 2.0.2 (Spark MLlib, Spark SQL, Spark Core)
- PySpark Framework
- Python version 2.7.2
- Bash scripting

**Description of measurements performed:**

For every part, we would analyze the impact of change in the number of executors on the performance of the code. The analysis for each part is shown below

**Part -2:** Execution time for CSV file format

| No. of Executors | 5 | 10 | 15 |
|---|---|---|---|
| Avg Time Taken (5 readings) (in seconds) | 932 | 847 | 563 |
| Min Time Taken | 878 | 820 | 511 |
| Max Time Taken | 988 | 763 | 672 |

Table 1: Time taken by CSV file format

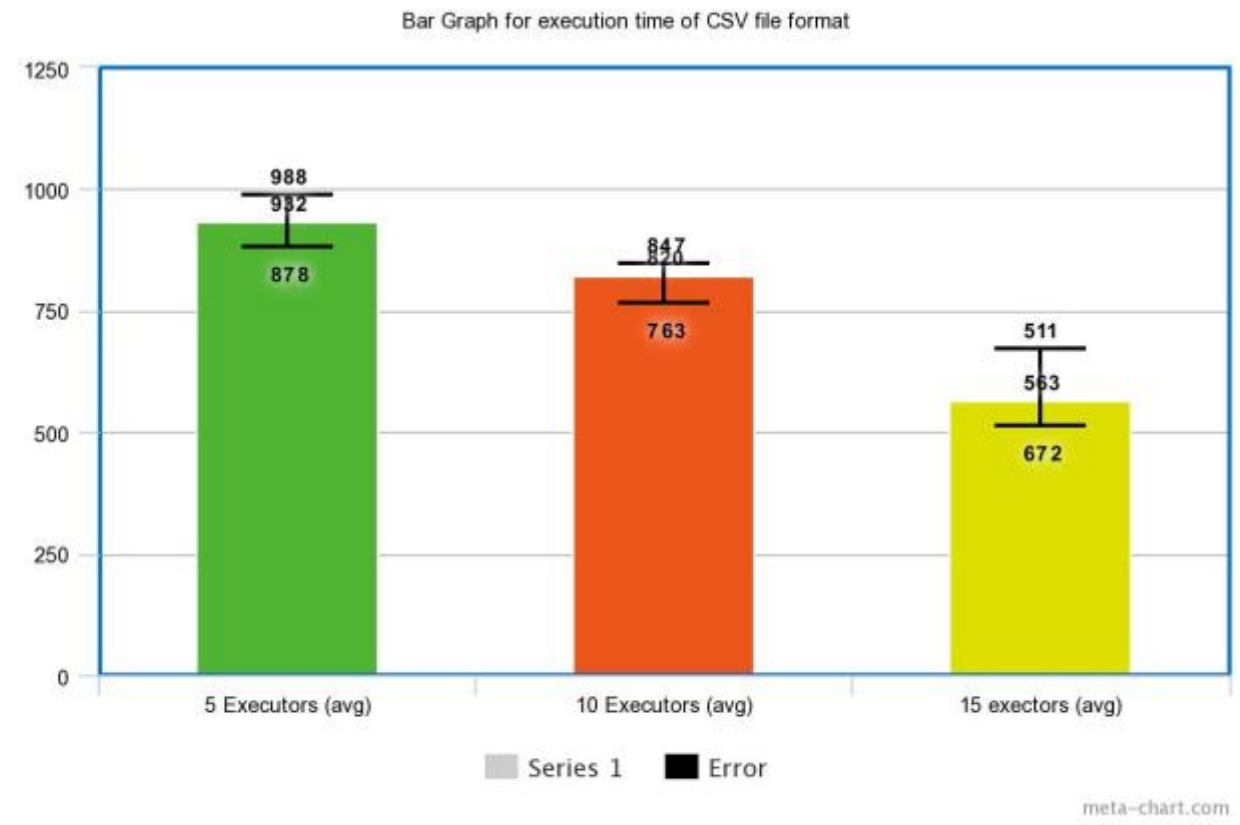

Fig 1. Time Taken by Part – 2 Execution time for CSV file format

**Part-2:** Execution time for JSON file format

| No. of Executors | 5 | 10 | 15 |
|---|---|---|---|
| Avg Time Taken (5 readings) (in seconds) | 1198 | 1052 | 972 |
| Min Time Taken | 1007 | 987 | 931 |
| Max Time Taken | 1440 | 1098 | 1004 |

Time Taken by Part -2 Execution time for JSON file format

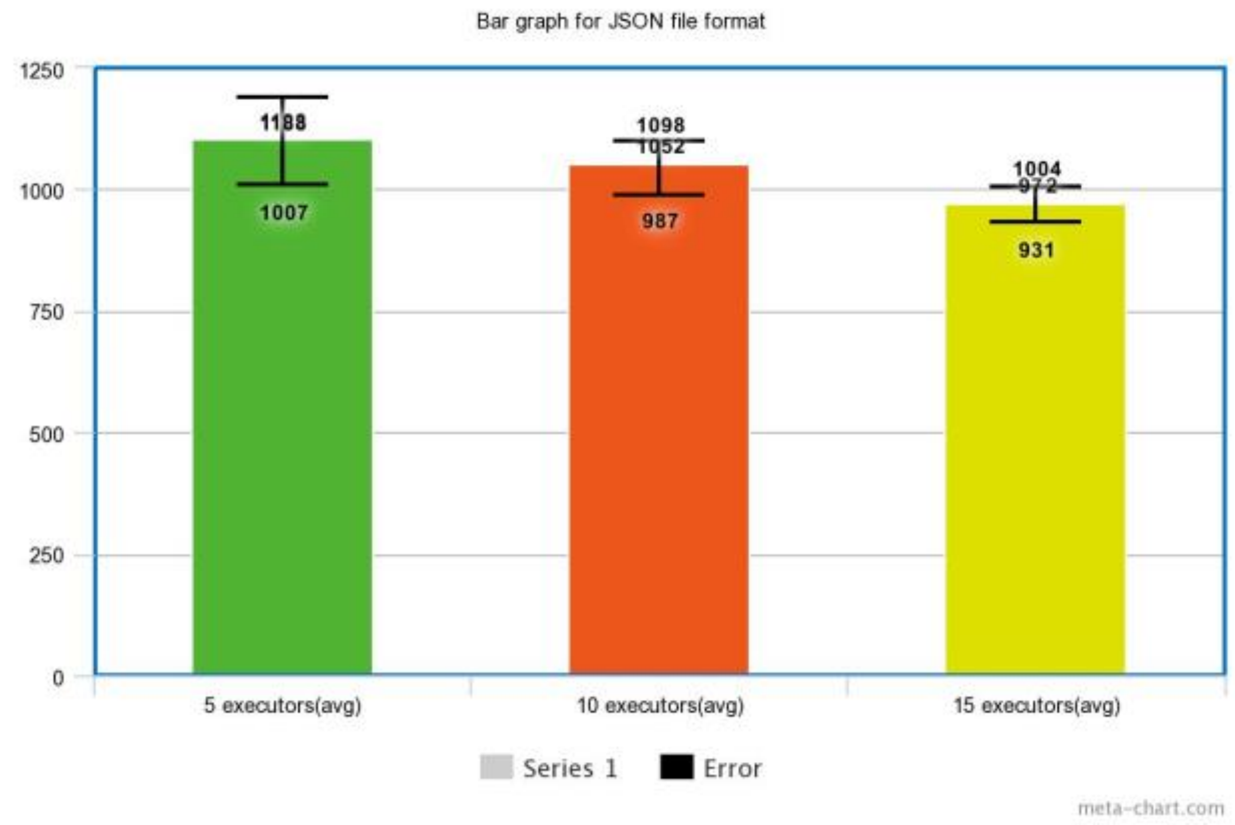

Fig 2. Time taken by Part – 2 Execution time for JSON file format

**Part-2** Execution time for Parquet file format

| No. of Executors | 5 | 10 | 15 |
|---|---|---|---|
| Avg Time Taken (5 readings) (in seconds) | 189 | 244 | 196 |
| Min Time Taken | 220 | 165 | 132 |
| Max Time Taken | 246 | 201 | 167 |

Time Taken by Part -2 Execution time for Parquet file format

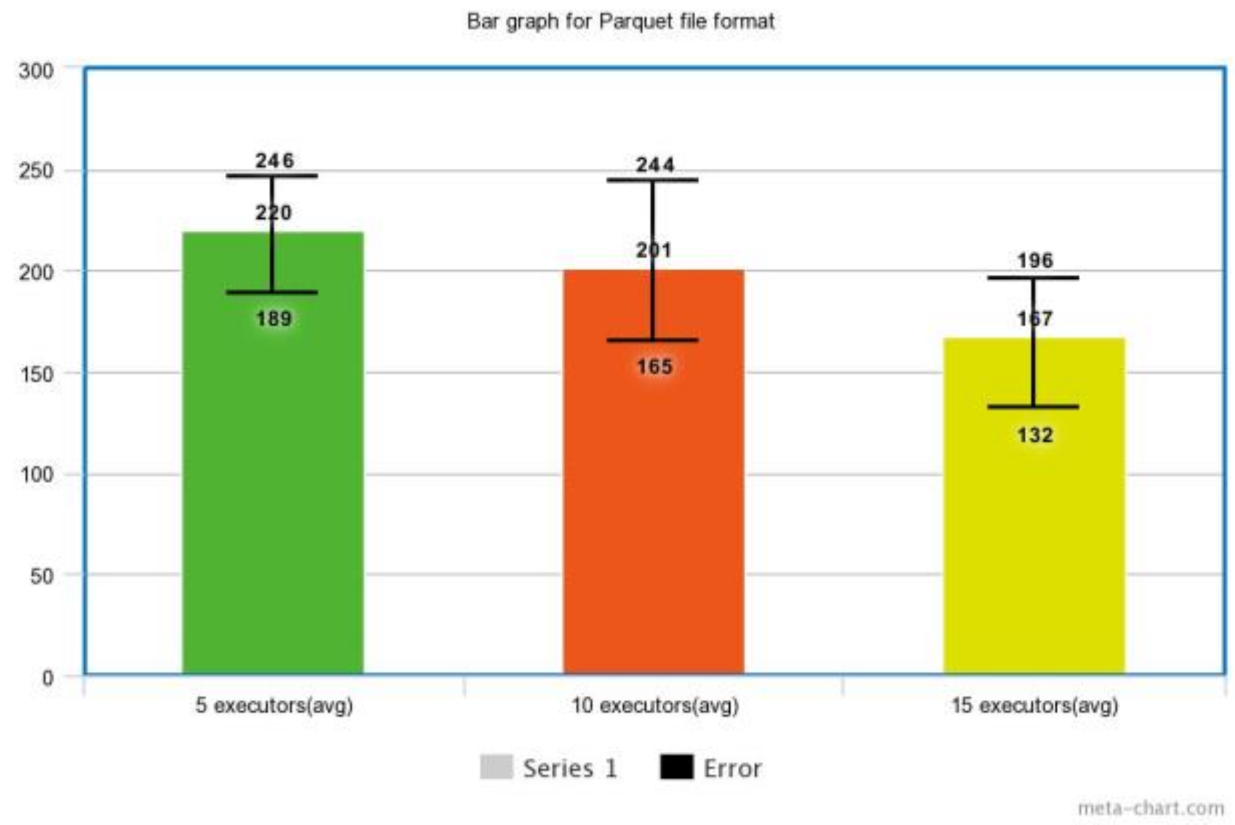

Fig 3. Time Taken by Part -2 Execution time for Parquet file format

**Part-2** Execution time for Sequence file format

| No. of Executors | 5 | 10 | 15 |
|---|---|---|---|
| Avg Time Taken (5 readings) (in seconds) | 504 | 485 | 467 |
| Min Time Taken | 488 | 412 | 411 |
| Max Time Taken | 586 | 567 | 504 |

Time Taken by Part -2 Execution time for Sequence file format

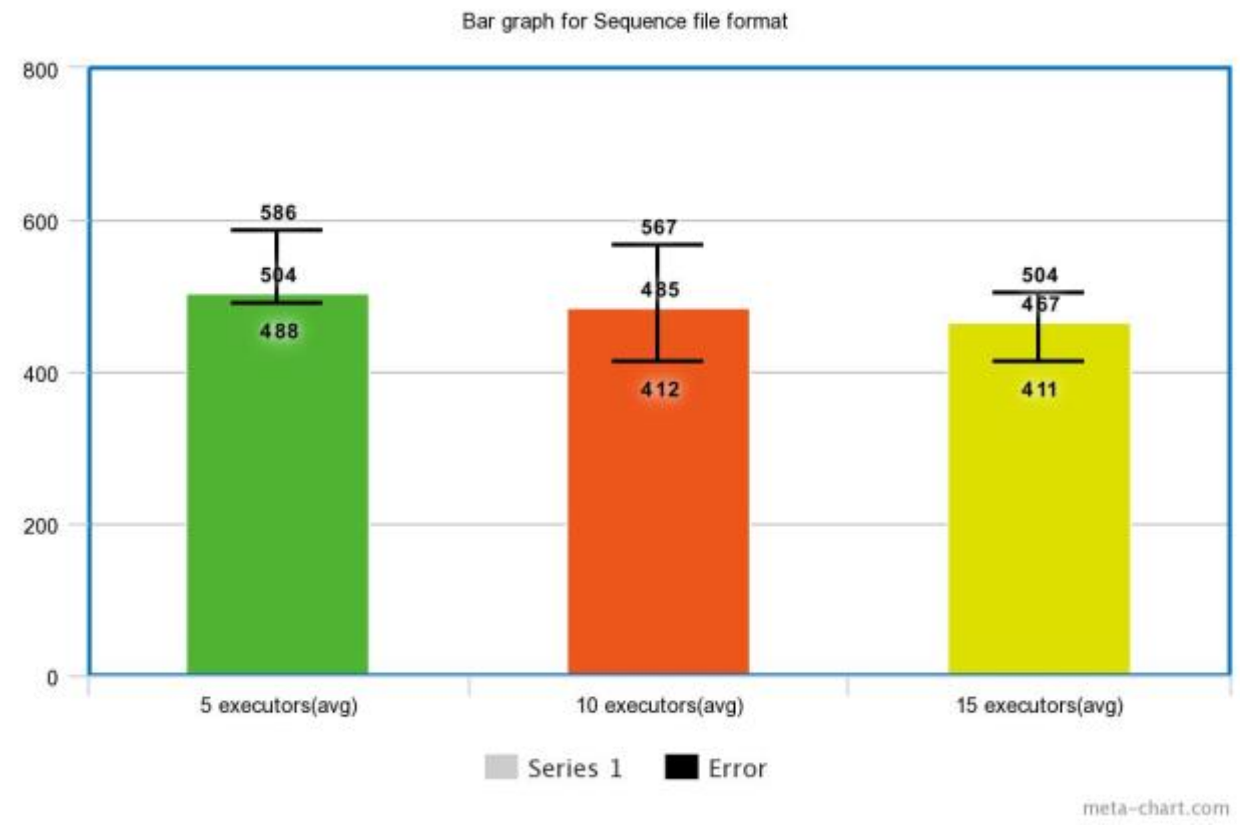

Fig 3. Time Taken by Part -2 Execution time for Sequence file format

**Findings:**

**Step 1:** In First step after converting the given CSV data to different file formats, following insights were obtained.

Note: To get the output directory size in HDFS I used below command

```
Hadoop fs –du –s –h <output directory path>
```

| File Format | Size in GB |
|---|---|
| CSV file format | 3.1 |
| JSON file format | 14.4 |
| Parquet file format | 560.9 |
| Sequence file format | 5.4 |

- Parquet file provided best compression compared to other file formats, it took 560.9 MB of space to store the 3.1 GB CSV data.
- Sequence File consumed more space than the original data it took 5.3 GB of space.
- JSON file format is the file format with very bad space efficiency it occupied 14.4 GB of disk space to store 3 GB of CSV data.
- The DataFrame which was again saved in the CSV file format after removing ‘NA’ values occupied same space as original file.

From the above outputs, we can conclude that Parquet file format is best for storing the text data when we should achieve optimal space efficiency.

Also, we can notice one more fact that Parquet file format being column oriented store provides better space efficiency than JSON and Sequence file formats which are key-value stores.

**Step 2**: In second step after executing the code to compute flight delay percentage per origin for different file formats, the following insights were obtained

- We have observed from the above tables and visualization the performance of parquet file format is several times faster than other file formats.
- Parquet file performs better since it used optimized compression algorithm to store the data which provides better read access performance compared to other file formats.
- Next to Parquet file is sequence file in terms of performance, since it also compresses the data, even though not as good as parquet file in terms of performance, but it’s still better than JSON and CSV file format.
- JSON file format performance is bad compared to another file format, since it directly stores the key value pairs without any compression.
- CSV file format even though performs better than JSON file format, it’s still way behind parquet file in terms of performance.

We can say that while handling text data it’s better to use Parquet file to achieve optimal performance.

Why we need to compute ideal configuration for any spark job?

Because this would prevent the starvation of other execution processes and helps in effective utilization of available resources.

By caching the RDD’s and DataFrames effectively we can improve the performance of the Spark Jobs. We can also persist the RDD’s and DataFrames onto disk also.

## Conclusion

Learnings from above assignment:

- The effective usage of Spark Cluster Computing model by leveraging on top of Hadoop distributed framework.
- An insight into different Hadoop supported file formats and their performance.
- The analysis of large data set to generate historical reports to make sense of the data.

Future Work:

- The above assignment can be extended to compute a predictive model such as Random Forest to forecast the flight delays by using the previous year’s flight data.
- Since the data contains other fields such as weather delay, NAS Delay, Security Delay etc., by performing further analysis we can identify which factor has frequently resulted in the delay of the flight.

## References:

1. http://pstl.cs.uh.edu/resources/whale