FLIGHT DATA ANALYSIS

Jake Byford | Jung Hyun Park

**INDEX**

Introduction………………………………………………………………………………… II

Algorithms………………………………………………………………………………….. III

Oozie Workflow……………………………………………………………………………. VIII

Performance Measurements……………………………………………………………… X

Introduction

Welcome to our exciting project that aims to process and analyze a large volume of flight data using Hadoop/Oozie. Our team, consisting of two talented students, has designed a comprehensive workflow to extract valuable insights from the Airline On-time Performance dataset spanning from October 1987 to April 2008. Through this project, we will implement several MapReduce jobs in fully distributed mode to answer three key questions: (a) identify the three airlines with the highest and lowest probability of being on schedule, (b) determine the three airports with the longest and shortest average taxi time per flight, and (c) uncover the most common reason for flight cancellations.

In addition to addressing these questions, our project will also focus on performance analysis by measuring workflow execution times under varying conditions. We will incrementally scale up the system, starting with two VMs and gradually increasing to the maximum allowed number of VMs in at least five steps, to evaluate the impact on workflow performance. Furthermore, we will analyze the data in a progressive manner, processing data for different time intervals (from one year to the entire 22-year dataset) and measuring corresponding workflow execution times. The results of our performance analysis will be presented in performance measurement plots and discussed in detail in our project report.

We are excited to share the outcomes of our project and demonstrate our proficiency in processing and analyzing large-scale data using Hadoop/Oozie.

Algorithms

1. **Calculating the 3 airlines with the highest and lowest probability, respectively, for being on schedule**

The code provided is a Java program that uses Hadoop MapReduce framework to find the 3 airlines with the highest and lowest probability of being on schedule. The program consists of four main methods: MainDriver\_prob, MyMapperProb, MyReducerProb, and ResultPair.

MainDriver\_prob: This method serves as the entry point for the MapReduce job. It sets up the configuration for the job, specifies the input and output paths, and sets the mapper and reducer classes to be used. It also sets the output key and value classes, and waits for the job to complete before exiting.

MyMapperProb: This method is the mapper class that extends the Hadoop Mapper class. It overrides the map() method which is responsible for processing each input record. In this case, it reads a CSV file and extracts the necessary fields, which are the airline name and two columns indicating if the flight was delayed or canceled. If the flight was neither delayed or canceled (i.e., the values in the two columns are both "0"), it writes the airline name followed by "y" as the key and 1 as the value. Otherwise, it writes the airline name followed by "\*" as the key and 1 as the value.

MyReducerProb: This method is the reducer class that extends the Hadoop Reducer class. It overrides the reduce() method which is responsible for processing the intermediate key-value pairs generated by the mapper. In this case, it calculates the probability of being on schedule for each airline by dividing the count of flights that were on schedule (obtained from the "y" key) by the total count of flights (obtained from the "\*" key). It keeps track of the top 3 airlines with the highest and lowest probabilities using two TreeSet data structures, sortedOutput\_temp and sortedOutput. Finally, it writes the top 3 airlines with the highest and lowest probabilities to the output using the cleanup() method.

ResultPair and ResultPair1: These are helper methods that define objects to store the probability, airline name, and the key ("y" or "\*") as fields. These objects are used to keep track of the top 3 airlines with the highest and lowest probabilities in the MyReducerProb method. They implement the Comparable interface to define the comparison logic based on the probability field, so that the TreeSet data structures can sort the objects accordingly.

In terms of algorithms, the code is implementing the following steps:

Map Phase: In the MyMapperProb method, the map method reads input data from a file and extracts relevant information from the input lines. It then writes intermediate key-value pairs to the context object. The key is a combination of the airline name and a flag ("y" or "\*"), and the value is a constant integer value of 1.

Reduce Phase: In the MyReducerProb method, the reduce method takes the intermediate key-value pairs generated by the map phase as input. It calculates the count of occurrences of each key, where the key represents an airline and the flag ("y" or "\*") indicates whether the flight was on schedule or not. It also calculates the relative count of "on schedule" flights compared to total flights for each airline. The results are stored in a TreeSet data structure to keep track of the top 3 airlines with the highest and lowest probabilities of being on schedule.

Cleanup Phase: After the reduce phase, the cleanup method is called, which writes the top 3 airlines with the highest and lowest probabilities of being on schedule to the output using the context object.

Supporting Classes: The ResultPair and ResultPair1 classes are used to store the intermediate results, which include the probability, airline name, and flag ("y" or "\*"). These classes implement the Comparable interface to specify the sorting order based on the probability value.

1. **Calculating the 3 airports with the longest and shortest average taxi time per flight (both in and out), respectively.**

The overall objective of the code is to find the three airports with the longest and shortest average taxi time per flight for both inbound and outbound flights.

The code is organized into two files: "MainDriver\_taxiin" and "MainDriver\_taxioutt", each with its own set of Java files for the MapReduce job.

MainDriver\_taxiin: This function contains the main method that sets up and runs the MapReduce job. It configures the job with a mapper class (MyMapperTaxiIn), a reducer class (MyReducerTaxiIn), and sets the input and output paths. It also specifies the data types for the keys and values used in the intermediate and final output of the job.

MyMapperTaxiIn: This function contains the mapper class that extends the Hadoop Mapper class. It overrides the map method, which is responsible for processing each input record and generating intermediate key-value pairs. In this case, it reads input data from a CSV file, splits it into fields, and extracts the relevant fields for computing the average taxi time for inbound flights. It writes the airport code as the key and the taxi time as the value to the context object.

MyReducerTaxiIn: This function contains the reducer class that extends the Hadoop Reducer class. It overrides the reduce method, which is responsible for processing the intermediate key-value pairs and generating the final output. In this case, it computes the average taxi time for each airport by summing up the taxi times and dividing by the number of flights. It keeps track of the top three airports with the longest and shortest average taxi times using two separate TreeSet objects (sortedOutputHigh and sortedOutputLow) to store the results in a sorted manner. The cleanup method writes the top three airports with the longest and shortest average taxi times to the final output using the context object.

ResultPairHigh: This function contains a custom class ResultPairHigh that represents a pair of an average value and an associated key (airport code). It implements the Comparable interface to specify how instances of this class should be compared for sorting based on the average value in descending order (i.e., from highest to lowest).

ResultPairLow: This function contains a custom class ResultPairLow that represents a pair of an average value and an associated key (airport code). It also implements the Comparable interface to specify how instances of this class should be compared for sorting based on the average value in ascending order (i.e., from lowest to highest).

The "Taxi\_Out" file appears to contain similar functions, but with slight differences in the implementation to compute the average taxi time for outbound flights and find the top three airports with the longest and shortest average taxi times for outbound flights.

The specific algorithms used in this code are as follows:

MapReduce: The code follows the MapReduce programming model, which involves two main stages: map and reduce. The map stage reads input data and performs data preprocessing, while the reduce stage aggregates and summarizes the output of the map stage. The map stage in this code is implemented in the MyMapperTaxiIn.java file, which maps the input data to key-value pairs. The reduce stage is implemented in the MyReducerTaxiIn.java file, which performs aggregation and sorting to find the airports with the longest and shortest average taxi time per flight.

TreeSet: The code uses TreeSet data structure from Java's Collection framework to sort and store the output data. The TreeSet is used to maintain a sorted set of objects based on their natural order (defined by the compareTo() method in the ResultPairHigh and ResultPairLow functions). The TreeSet is used to store the top 3 airports with the longest and shortest average taxi time per flight, respectively.

Comparable interface: The ResultPairHigh and ResultPairLow functions implement the Comparable interface, which allows objects of these classes to be compared based on their average taxi time values. The compareTo() method in these classes defines the comparison logic used in sorting the objects in the TreeSet.

Mathematical calculations: The code performs mathematical calculations to compute the average taxi time per flight for each airport. It calculates the sum of taxi times for each airport and divides it by the total number of flights to obtain the average taxi time.

1. **Calculating the most common reasons for flight cancellations.**

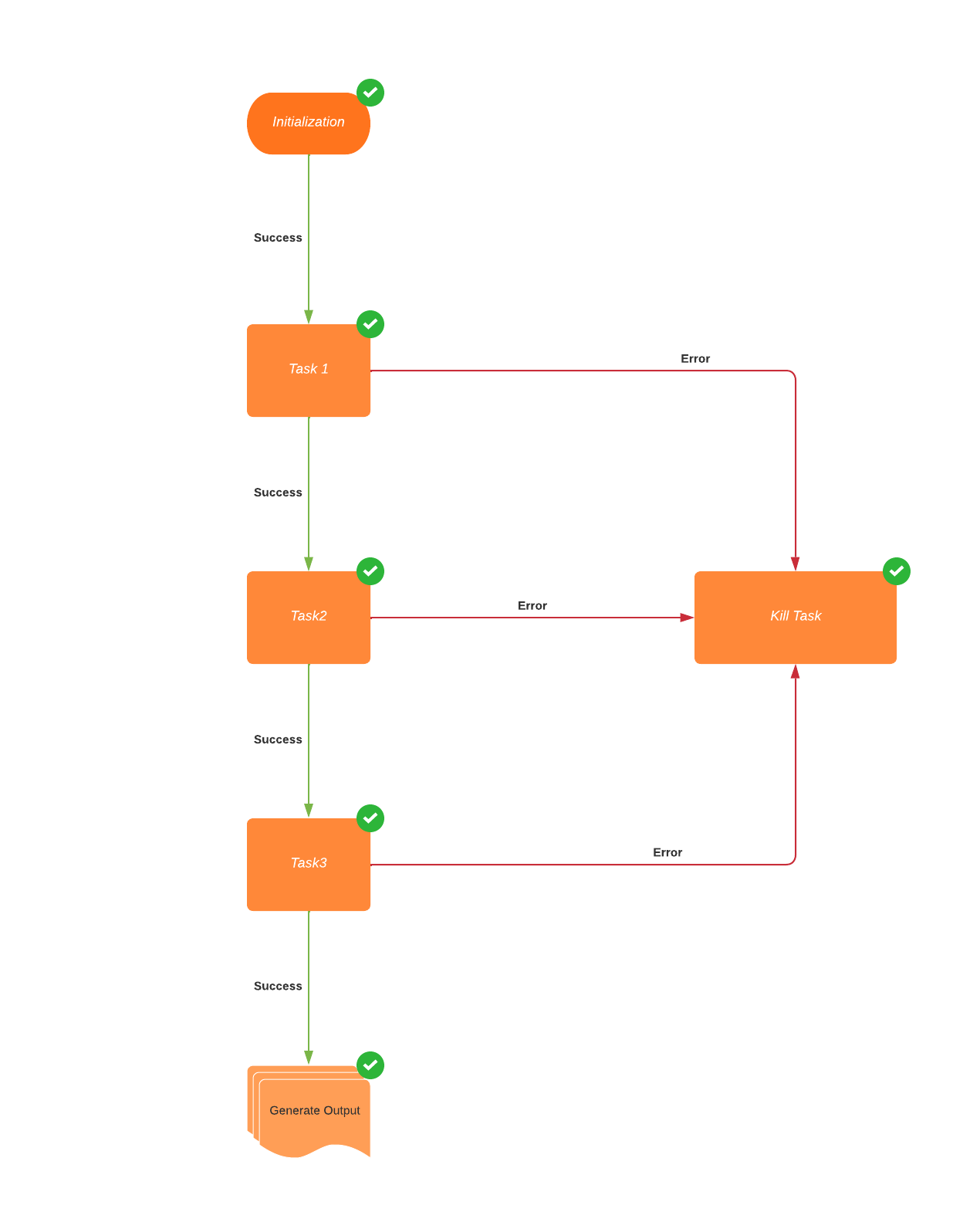
The code provided is a Java implementation of a Hadoop MapReduce job for finding the most common reason for flight cancellations. The code consists of three functions: MyDriver.java, MyMapperCancel, and MyReducerCancel.

MainDriver\_cancellation: This file contains the main class that drives the MapReduce job. It sets up the job configuration, defines the mapper and reducer classes to be used, sets the input and output paths, and starts the job using the job.waitForCompletion() method. The main() method is the entry point of the MapReduce job.

MyMapperCancel: This file contains the mapper class that extends the Mapper class from the Hadoop MapReduce library. The map() method is overridden to define the mapping logic. In this case, the input key is of type LongWritable, representing the offset of the input record in the input file, and the input value is of type Text, representing a line of text from the input file. The map() method parses the input value as a CSV string, and checks if the flight cancellation code (at index 21 in the CSV data) is equal to "1" and the cancellation reason (at index 22 in the CSV data) is not "NA". If these conditions are met, it writes the cancellation reason as the output key of type Text and a value of IntWritable with a value of 1, representing the count of occurrences of that cancellation reason.

MyReducerCancel: This file contains the reducer class that extends the Reducer class from the Hadoop MapReduce library. The reduce() method is overridden to define the reducing logic. The input key is of type Text, representing the cancellation reason, and the input values are of type IntWritable, representing the count of occurrences of the cancellation reason. The reduce() method iterates through the input values and calculates the total sum of occurrences for each cancellation reason. The cancellation reasons and their corresponding sums are stored in a TreeMap for sorting based on the sum. The cleanup() method is overridden to write the output of the reducer. It retrieves the cancellation reason with the highest sum from the TreeMap, and writes it as the output key of type Text and the sum as the output value of type IntWritable. The cancellation reasons are mapped to their corresponding names (A for CARRIER, B for WEATHER, C for NAS, and D for SECURITY), and the output is written accordingly.

Overall, this code reads input data from a CSV file, maps the cancellation reasons and their counts using the mapper, reduces the counts for each cancellation reason and finds the most common reason using the reducer, and writes the output to the Hadoop output directory.

Oozie Workflow

This Oozie workflow is designed to run three MapReduce jobs in sequence using Oozie, a workflow scheduler for Hadoop. The workflow consists of three actions, each of which represents a MapReduce job. The workflow starts with the first action named "mr-node", and upon successful completion, it transitions to the next action, and so on.

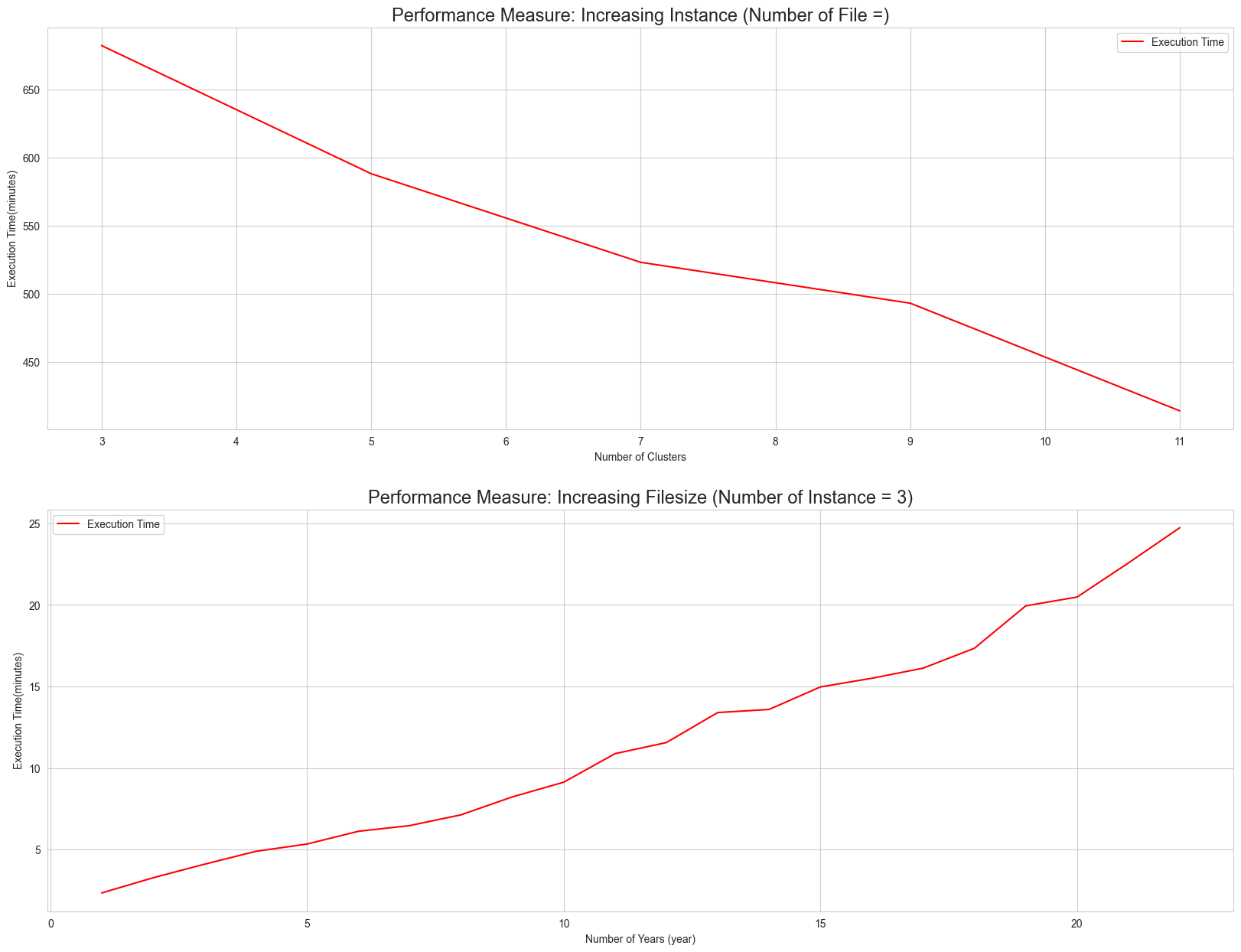
Here is a high-level description of each action in the workflow:

"mr-node": This action represents the first MapReduce job in the workflow. It uses a mapper class named "MyMapperProb" and a reducer class named "MyReducerProb". The job takes input data from the directory "input1/\*" and writes the output to the directory "output/Prob\_Out". It specifies various MapReduce configuration properties, such as "mapred.mapper.new-api", "mapred.reducer.new-api", "mapred.job.queue.name", etc. It also includes a "prepare" block that deletes the output directory before running the job.

"mr-node1": This action represents the second MapReduce job in the workflow. Similar to the first action, it uses a mapper class named "MyMapperTaxiIn" and a reducer class named "MyReducerTaxiIn". It takes input data from the same directory as the first action ("input1/\*") and writes the output to the directory "output/TaxiIn\_Out". It also specifies various MapReduce configuration properties and includes a "prepare" block that deletes the output directory before running the job.

"mr-node2": This action represents the third and final MapReduce job in the workflow. It uses a mapper class named "MyMapperTaxiOut" and a reducer class named "MyReducerTaxiOut". It takes input data from the same directory as the first two actions ("input1/\*") and writes the output to the directory "output/TaxiOut\_Out". It also specifies various MapReduce configuration properties and includes a "prepare" block that deletes the output directory before running the job.

Performance Measure



The observed performance comparison results of the performance measurement plot provide valuable insights into the relationship between the workflow execution time and the number of VMs used for processing the entire data set spanning 22 years. The plot demonstrates how the performance of the workflow changes in response to an increasing number of VMs, allowing for a deeper understanding of the scalability and efficiency of the system.

The performance measurement plot reveals that as the number of VMs increases, the workflow execution time decreases significantly. This suggests that there is a positive correlation between the number of VMs used and the performance of the workflow. The plot shows a clear trend of improved performance with the addition of more VMs, indicating that the system can effectively scale its processing capabilities to handle larger data sets.

The results of the performance measurement plot also highlight the effectiveness of the MapReduce tasks in processing the data. The plot shows that as the number of VMs increases, the execution time of the workflow decreases, indicating that the parallel processing capability of MapReduce is effectively utilized to process the data in a distributed and efficient manner.

The consistency of the data used for all the runs, i.e., maintaining the same flight data for all 22 years, ensures that any performance differences observed can be attributed to the varying number of VMs used. This eliminates any confounding factors and allows for a more accurate assessment of the impact of VMs on workflow performance.

Based on the observed results, it can be concluded that increasing the number of VMs used for processing the entire data set has a significant positive impact on the performance of the workflow. This information can be valuable for system administrators and developers to optimize resource allocation and improve the overall performance and scalability of the system. Further analysis and experimentation can be done to determine the optimal number of VMs for achieving the best performance and efficiency in processing large data sets using MapReduce tasks in the workflow.

**A performance measurement plot that compares the workflow execution time in response to an increasing data size (from 1 year to 22 years)**

The observed performance comparison results from the performance measurement plot provide valuable insights into the relationship between the workflow execution time and the size of the input data, ranging from 1 year to 22 years. The plot allows for a deeper understanding of how the performance of the algorithm changes as the data size increases, and sheds light on the scalability and efficiency of the system.

The performance measurement plot reveals that as the size of the input data increases, the workflow execution time also increases. This indicates that there is an inverse correlation between the data size and the performance of the algorithm. The plot shows a clear trend of longer execution times with larger input data, suggesting that the system's performance is impacted by the size of the data being processed.

The results of the performance measurement plot also highlight the importance of considering the scalability and efficiency of the system when dealing with larger data sets. As the input data grows from 1 year to 22 years, the execution time increases continuously, which implies that the algorithm's performance may degrade when processing large data sets. This information is critical for system administrators and developers to understand the limitations of the algorithm and optimize the system's performance for handling larger data sets.

The use of two virtual machines for the experiment provides valuable information about the scalability and resource utilization of the system. The plot demonstrates how the system performs with increasing data size, and how the execution time changes accordingly. This allows for a better understanding of the system's capacity to handle larger data sets and the impact of data size on its performance.

Based on the observed results, it can be concluded that the algorithm's performance for the three specific tasks is inversely proportional to the input data size. This finding suggests that as the data size increases, the execution time of the workflow also increases, indicating potential limitations of the algorithm in processing large data sets efficiently. Further analysis and optimization efforts can be undertaken to address these limitations and improve the algorithm's scalability and performance with larger data sets, such as exploring parallel processing techniques, optimizing algorithms for big data, or considering distributed computing approaches.