Flight Data Analysis

CS 644 Introduction to Big Data

Fall 2017

By

Clinton Pinto (cp359)
Shubham Gulia (sg952)

Clinton Pinto
Shubham Gulia

Index

1. Introduction

The project conducts a detailed analysis of the 22 year Airline Data using 3 map reduce programs running in a sequence using Oozie workflow. The MapReduce programs are run in fully distributed modes on 2-7 node clusters.

We created and used AWS instances to run the MapReduce jobs. The large volumes of data were analyzed and the oozie workflow was developed to find out the following data:

a) The 3 airlines with the highest and lowest probability, respectively, for being on schedule
b) The 3 airports with the longest and shortest average taxi time per flight (both in and out) respectively
c) The most common reason for flight cancellations

The project analyzes the entire data set (total 22 years from 1987 to 2008) at one time on two VMs first and then gradually increase the system scale to 7 Virtual machines (5 incremental steps) and each corresponding workflow execution time was measured.

The workflow was also used to analyze the data in a progressive manner with an increment of 1 year, i.e. the first year (1987), the first 2 years (1987-1988), the first 3 years (1987-1989) and the total 22 years (1987-2008), on the maximum amount of VMs i.e 7, and each corresponding workflow execution time was measured.

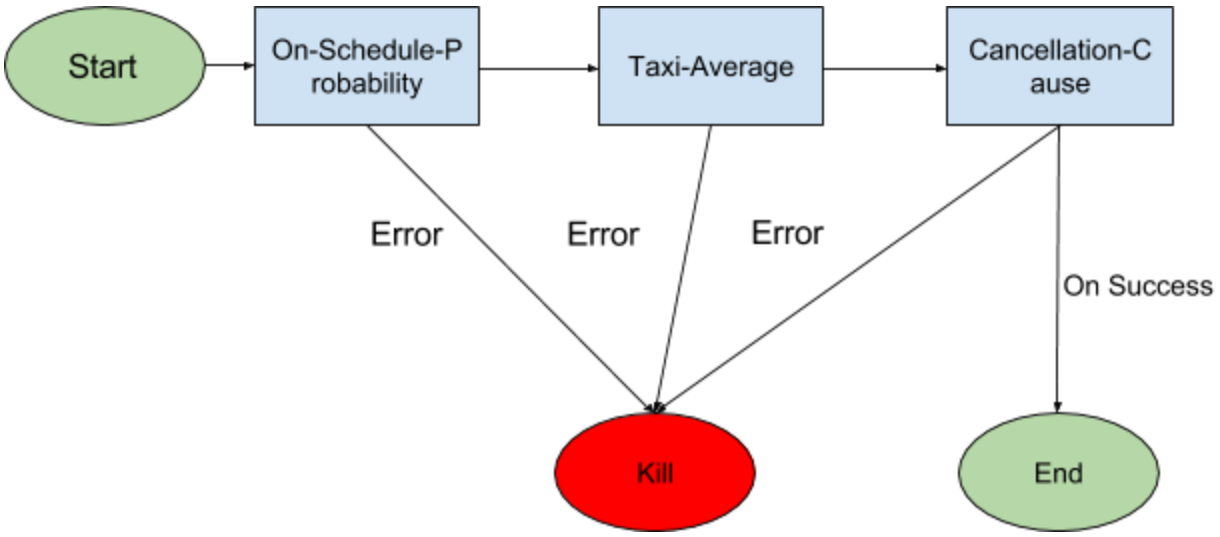
Tools Used

- Amazon Web Services (AWS) EC2 Ubuntu instances
- Apache Hadoop 2.8.2 (Fully Distributed mode)
- Apache Oozie
- Eclipse IDE - For writing the MapReduce programs and creating JARs

Clinton Pinto
Shubham Gulia

2. Oozie Workflow

The following figure gives a detailed overview of our oozie workflow. It consists of three MapReduce jobs running in sequence.



The entire workflow consists of three MapReduce jobs. On the success of the first job, the second job is started. On the success of the second job, the third job is started.

a) The oozie workflow starts off with the On-Schedule-Probability job which analyzes the data to find the top 3 airlines having the lowest probability of being on schedule and the top 3 airlines with the highest probability of being on schedule. On success of this job, the oozie workflow sends an ok to the next job to execute. In case of an error, the process is killed.
b) When the workflow gets an ok from the first job, the second job i.e. Taxi-Average is started. This job computes the 3 airports with the longest average taxi time and the 3 airports with the shortest average taxi time. On success of this job, the workflow will give an ok to the next job. In case the current job encounters an error, it kills the process.
c) On getting an ok from the previous task, the third job begins execution. The Cancellation-Cause job parses the data and finds out the most common cause for cancellation. It logs the cancellation code of the most common reason. On completion of this task, the workflow process was a success and the jobs are all completed. The output is available in the folders output1, output2 and output3.

3. MapReduce Algorithms

3.1. On-Schedule-Probability

3.1.1. Mapper (AirlineDelayProbabilityMapper.java)

1) To find the probability, we consider a flight is late if it takes off 5 minutes after its scheduled departure time and arrives 5 mins after its scheduled departure times. Its safe to assume a limit of 10 mins of delay. We initialize the MAXIMUM_DELAY_LIMIT to 10 mins.
2) Parse the line to get the year, the carrier, the departure delay and the arrival delay
3) Check if the entry is valid, if valid go to point 4. Else go to point 7.
4) If sum of departureDelay and ArrivalDelay is greater than MAXIMUM_DELAY_LIMIT, go to point 5. Else go to point 6.
5) Write the mapper output with the carrier as the key and 0 as the IntWritable value
6) Write the mapper output with the carrier as the key and 1 as the IntWritable value
7) Repeat 2-6 for all input lines
8) END

3.1.2. Reducer (AirlineDelayProbabilityReducer.java)

1) At setup of reducer task, initialize the list (onTimeFlightList) which holds objects FlightOnTime, which contains the airline and its probability for being on schedule.
2) Initialize the onTimeFlightCounter to 0. Initialize the totalCount to 0.
3) For each carrier key values, repeat 4,5.
4) If the value is 1, increment onTimeFlightCounter
5) Increment the totalCount
6) Create new FlightOnTime object with key as airline name and probability by calculating probability using onTimeFlightCounter / totalCount
7) Add new FlightOnTime object to list onTimeFlightList
8) Repeat 2-7 for all output key-values received from mapper
9) On completion of reduce task, check if list is empty. If list is empty, go to 13
10) Sort the onTimeFlightList in ascending order of probabilities.
11) Log the first three elements of the list as reducer output. (Airline and probability) with the title “the airlines with the lowest probability”
12) Log the last three elements of the list as reducer output (Airline and probability) with the title “the airlines with highest probability”
13) END

3.2. Taxi-Average

3.2.1 Mapper (AirportTaxiTimeMapper.java)

1) Parse the line to get the origin, the destination, the taxi in time and the taxi out time.
2) Check if the entry is valid, if valid go to point 3. Else go to point 7.
3) Write the origin airport as the key and the taxi out time as the value for mapper output.
4) Write the destination airport as the key and the taxi in time as the value for mapper output.
5) Repeat 1-4 for all input lines.
6) END

3.2.2 Reducer (AirportTaxiTimeReducer. java)

1) At setup of reducer task, initialize the list (airportsList) which holds objects Airport, which contains the airport code and its average taxi time.
2) Initialize the totalTaxiTime to 0. Initialize the totalTimesTaxied to 0.
3) For each carrier key values, repeat 4,5.
4) totalTaxiTime = totalTaxiTime + value
5) Increment the totalTimesTaxied
6) End for
7) Calculate average time for this carrier key using totalTaxiTime / totalTimesTaxied
8) Create new airport object with key as airport name and average time
9) Add new Airport object to list airportsList
10) Repeat 2-9 for all outputs received from mapper
11) On completion of reduce task, check if list is empty. If list is empty, go to 15
12) Sort the airportsList in ascending order of average time of each airport.
13) Log the first three elements of the list as reducer output. (Airport and average time) with the title “the airport with the lowest average taxi time”.
14) Log the last three elements of the list as reducer output (Airport & Average Taxi Time) with the title “the airport with highest average taxi time”
15) END

3.3. Cancellation-Cause

3.3.1 Mapper (CancellationCauseMapper.java)

1) Parse the line to get the isCancelled and cancellation Reason code
2) Check if the entry is valid, if valid go to point 3. Else go to point 4.
3) Write the cancellation Reason code as mapper output key and 1 as the IntWritable value
4) Repeat 1-3 for all input lines.
5) END

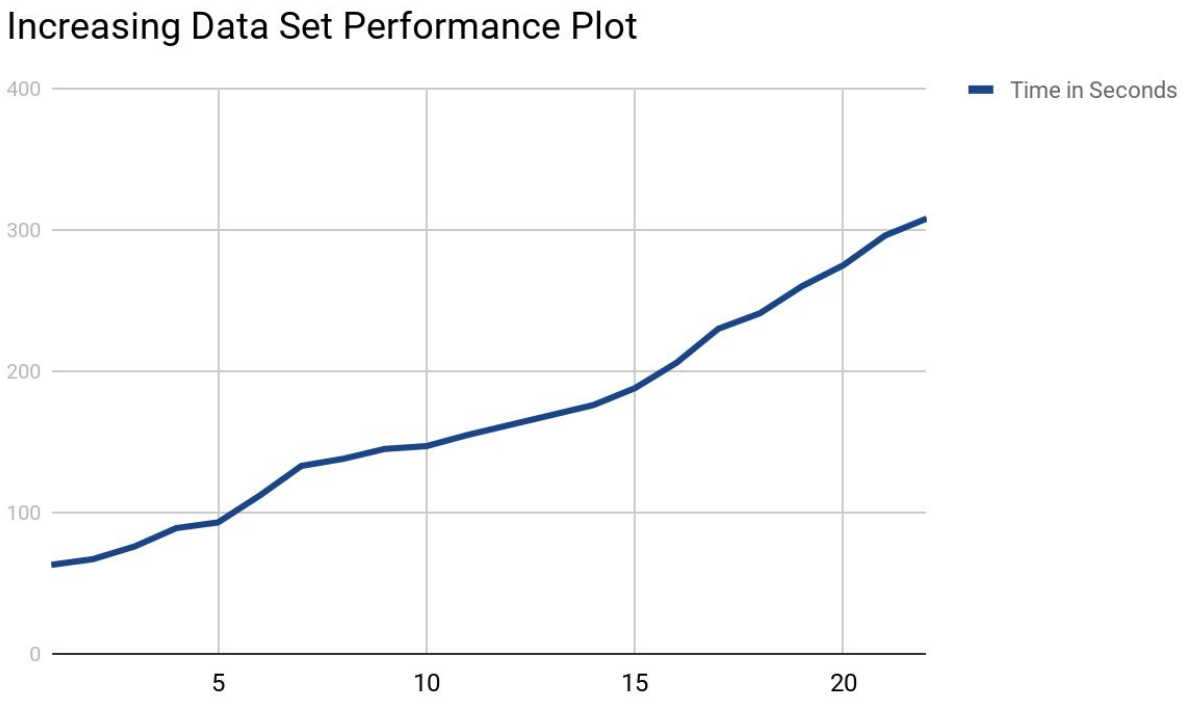
3.3.2 Reducer (CancellationCauseReducer.java)

1) At setup of reducer task, initialize the list (cancellationList) which holds objects cancelInfo, which contains the cancellationCode and its count.
2) Initialize the counter to 0
3) For all values for current key, the counter = counter + value
4) Create new object cancelInfo, with current cancellationCode as key and its count as counter.
5) Add the cancelInfo object to the cancellationList.
6) Repeat 2-5 for all keys
7) On completion of Reduce task, check if cancellationList is empty. Print “No data available” as output of reducer. Go to 10.
8) Sort the cancellationList in ascending order of counts
9) Output the cancellation key and the count of the last object in the list as the reducer output.
10) END

Clinton Pinto
Shubham Gulia

4. Performance Measurement Plot - I

- Increasing Data Set



Incremental Data in years (1-22)

We measured the performance of the system by increasing the data set incrementally, starting from 1987 to 2008. The execution time of the system was recorded for each year. The maximum number of virtual machines i.e. 7 were used for this performance analysis.

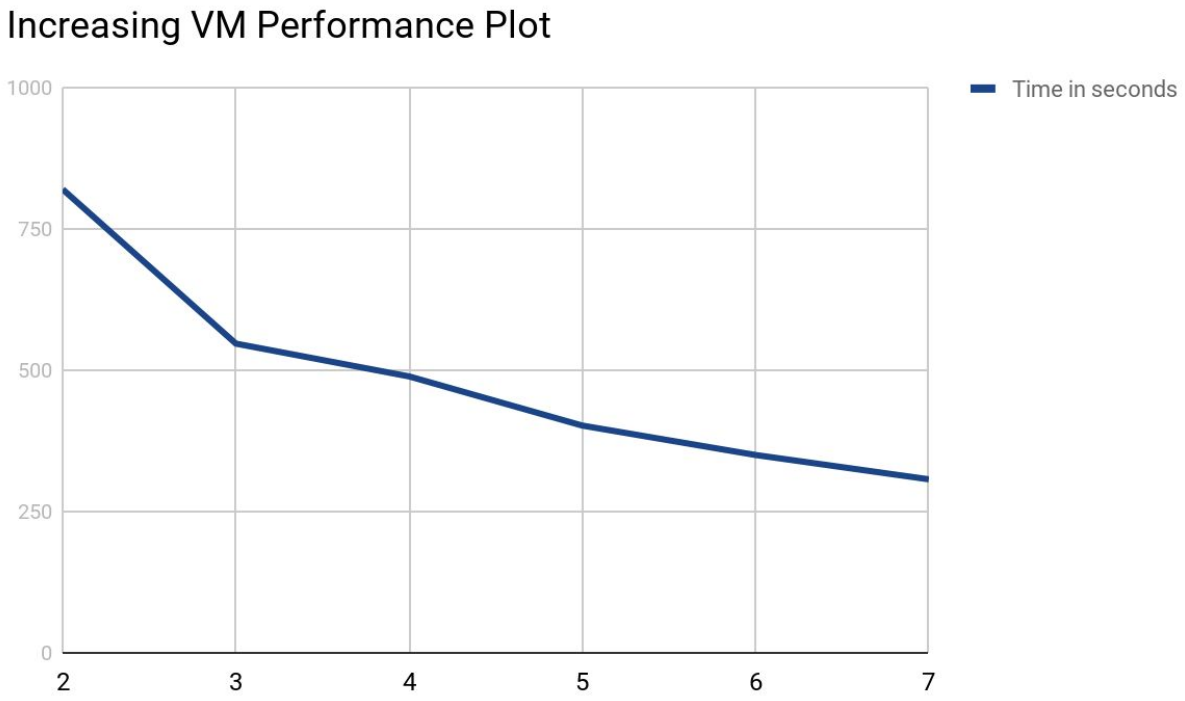
We started with the single year 1987. It took approximately 63 seconds for the system to complete the execution. We kept increasing the data set in steps of one year. For eg. 1987, 1987-1988, 1987-1989, 1987-1990 …… 1987-2008. As you can see in the graph, as the data size increased, the amount of time to process the data also increased. The execution time increased from 63 seconds to 308 seconds between the data sizes of the year 1987 and 1987-2008 respectively. This shows that there was an increase by a factor of approximately 4.8.

Since the computational resources stay the same. The increasing data size will require more processing time to compute the final output. This performance behavior is definitely what was expected from the system.

Clinton Pinto
Shubham Gulia

5. Performance Measurement Plot - II

- Increasing number of Virtual Machines



In this performance analysis, the data size was kept constant at 22 years. However the data was analyzed by tweaking the number of virtual machines used for processing the data.

We began the performance analysis by processing the entire data set of 22 years only on 2 Virtual machines. We gradually kept adding virtual machines and noted down the execution time. For the cluster with two virtual machines, the execution time was approximately 820 seconds. For the cluster with 7 virtual machines, the execution time was approximately 307 seconds. As seen in the graph, the execution time drops as the number of virtual machines is increased.

This behavior was expected as the data size remains constant and the number of virtual machines in the cluster is increased. The execution time decrease by a factor of approximately 2.7. This is textbook behavior as the processing speed increases as the number of virtual machines increases.