12-2016

Network Log Analysis Performance Comparison - Java vs. MapReduce

Jonathan C. Munsch
jcmunsch@stcloudstate.edu

Network Log Analysis Performance Comparison:

Clustered MapReduce Verses Custom Java

by

Jonathan Munsch

A Thesis

Submitted to the Graduate Faculty of

St Cloud State University

in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in Information Assurance

November, 2016

Thesis Committee:

Dr. Dennis Guster, Chairperson
Dr. Jim Chen
Dr. Mark Schmidt

Abstract

The goal of this thesis is to establish a benchmark comparison of custom Java based code efficiency as it relates to similar MapReduce jobs. Four separate tasks were completed with custom Java and MapReduce code to produce the identical output. Network pcap data was analyzed with tshark, and the resulting text file used as input for the programs to be run. Each code base was required to determine the following information from the tshark data: a summation of the number of port access attempts by source IP address, the total traffic volume by IP protocol, the average packet length by source IP address, and the percentage of traffic volume by source IP address. All tests were performed within an Amazon Web Services environment, and multiple test runs were executed to ensure the overall efficiency was not affected by possible shared resources. A cost-benefit analysis was performed to determine a point in which MapReduce and Hadoop clusters are worth the extra cost of additional hardware based upon the cost comparison of one AWS EC2 instance versus a four cluster HDFS system.

Acknowledgments

I would like to thank my wife, Andrea Munsch for all her support and guidance during the entire thesis process. I could not have completed this without her love, encouragement, and advice.

TABLE OF CONTENTS

Appendix

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Chapter I

INTRODUCTION

Introduction

According to Cisco's 2015 Global Cloud Index, by 2019 global data center traffic will have reached 10.4 zettabytes per year, with 83 percent of the data center traffic coming from the cloud (Global Cloud Index, 2015). TCP, UDP, FTP, SSH, and any new protocols that are likely to be created in the next three years will be utilized by an influx of new devices as the Internet of Things takes over homes and the "Bring Your Own Device" movement takes over corporations. In order to keep up with the massive influx of data, this paper has explored the utilization of custom java programing to provide an alternative to existing Map Reduction processes within Hadoop-based Big Data searches. In addition to a computational analysis discussing the benefits and limitations of each method, a cost-benefit analysis was created to determine which size data set to utilize a HDFS clustered environment verses a single system setup.

Problem Statement

As the traffic into networked information technology systems increases, the ability to discern a valid user from one who has malicious intent is becoming impossible to manage within current log-based intrusion detection methods. New ways to protect

systems must be discovered to separate valid and invalid access and rapidly analyze log data.

Nature and Significance of the Problem

As the use of cloud computing services grows, the personal information of individuals worldwide will reside within a cloud-based system. In order to protect the systems from attacks, external and internal, information security professionals will need to quickly and accurately obtain information from every access point within their networks. Intrusion Detection Systems (IDS) will need to advance their strengths in data analysis in order to prevent attacks in real time or remove them from a system before any damage can be done. This will include appropriately dealing with the multitude of devices migrating into networks due to the Internet of Things, flooding bandwidth with inexpressible possibilities for malicious intent to be carried out. Each innovation and new cloud-based device brought to market forces security professionals to contend with expanding access points for intrusion into their systems, creating an expanding volume of data that needs to be analyzed expeditiously.

Objective of the Study

The initial objective of this study was to provide a comparison of Graphic Processing Units versus Central Processing Units in relation to their performance ability when analyzing network log data through MapReduce queries. Due to limitations with the intended procedure, this process was modified to a comparison of custom Java

code against MapReduce processes run on network log data. The research performed a side by side comparison to determine any benefits of processing the analysis of said data without the use of clustered Hadoop-based systems and determined specific scenarios for the usage of each technology.

Research Questions/Hypotheses

The following questions were proposed prior to the research being completed within this project:

1) What is the performance increase in utilizing native Java versus clustered MapReduce on varying sized data sets?
2) What is the specific data set size MapReduce provides enough benefit to warrant the extra cost of the hardware?
3) Can cloud computing CPU clusters be utilized to offset the higher cost of hardware-based clustered services?

Limitations of the Research

Initial difficulties emerged within this study due to two issues. The first was the non-standard nature of PCAP files. Because packet sizes vary upon the information being processed, the size of each one can be quite varied. This caused complications when attempting to run parallel processes over the data as it needed to be structured first. Initial attempts were made to utilize the Hadoop-pcap library to process the data.

This proved problematic, and to work around the issue tshark was utilized to move the data out of pcap format and into a text format.

The second issue that was discovered related to difficulties of Aparapi code within a MapReduce function. Initial plans were to perform the actual map functionality across a GPU to exponentially increase efficiency. However, due to the design of the map class, this would have resulted in sending one process to the GPU at a time, providing no gains. To counteract this, research was moved into a more direct comparison of Hadoop MapReduce technology versus GPU based systems. The intended results being that the parallel processing that is being done over multiple clusters by a MapReduce job, can be performed on one GPU based system with a lower hardware footprint, and increased efficiency. Difficulties emerged with this process, as the current implementation of the Aparapi API used to quickly process Java code through a GPU does not currently allow the use of certain primitive variable types, most notably, the String type. The usage of String types within Aparapi is currently in the process of being added to the API with version 8 of the Java SDK and could provide a testbed for a future expansion on ideas within this paper in the future.

Definition of Terms

Graphical Processing Unit (GPU): Specially designed processor that is structured to process information in a highly parallel structure. Excels at performing multiple similar operations simultaneously.

Central Processing Unit (CPU): Specially designed circuit that performs the basic core function of a computer system.

Big Data: Term utilized to describe data sets that have reached a size that makes them unable to be processed by traditional techniques. The term focuses on the volume, variety, and velocity of data, meaning that it deals with large data sets that contain information from multiple sources that changes rapidly.

Bring Your Own Device (BYOD): Term that describes the current movement for employees to be able to utilize their own personal phones, tablets, laptops and other computing devices within the work environment.

Cloud Computing: Network-based systems that utilize pooled resources to provide services (IaaS, SaaS, and PaaS) to be accessed and requested on demand.

Hadoop: Open source software that utilizes a parallel file system that disperses processing and storage across multiple system clusters. It works heavily with MapReduce and is utilized to combat the problems generated by Big Data.

Internet of Things (IOT): Term utilized to describe the network of physical devices that contains network connectivity embedded within. Examples of such devices are WiFi enabled vehicles, WiFi enabled lighting systems, doorbell cameras, and web cameras.

Intrusion Detection System: Hardware or software device intended to analyze network activity for malicious intent.

MapReduce: Programming model utilized with Big Data related frameworks. In essence, it utilizes two separate functions to first filter and sort data, or maps the data, and then summarizes the findings or reduces the data.

Parallel Processing: A style of computational processing that converts large processes into smaller parts that are executed simultaneously.

Zettabyte: A message of storage capacity. One zettabyte is approximately one billion terabytes.

Summary

The previous chapter established the danger that current systems will face in the near future as a result of the inability to properly analyze network log information. Recent advances in technological areas have produced a scenario where more data is being generated and stored than ever before, and can easily be accessed at any given moment from any location. The next chapter will further explore the areas of Big Data, how the influx of network log information is pushing the current Intrusion detection systems to their limits, and how the utilization of GPU processing could be the solution to ensure any dataset can be analyzed.

Chapter II

BACKGROUND AND REVIEW OF LITERATURE

Introduction

This chapter identified problems attributable to the growth in network activity worldwide by providing an in-depth look at the services and systems responsible for the massive influx of network traffic. Additionally, current research and experimentation attempts to increase the functionality of current Intrusion Detection Systems and log analysis were reviewed, pinpointing possible areas for future improvement. The chapter will conclude with a discussion of how tools such as Hadoop, MapReduce, Aparapi and GPU processing can be used in unison to create a scalable framework that will allow for logs of any size to be quickly and accurately analyzed.

Background Related to the Problem

In order to sift through the flood of information that is bombarding current networks, it is imperative that network administrators understand the source. Recall that Cisco (2015) in their GCI report for 2014 – 2019 stated that 83 percent of all data center traffic will be coming from the cloud with the amount of cloud data increasing from 2.1 zettabytes to 8.6 zettabytes. The technology currently has a high impact on network activity, but with this suggested growth, it can clearly be seen as the root of the data problem. This massive amount of data stems from the very nature of the cloud computing service. Mell and Grance (2011) defined cloud computing as a way to

provide on-demand network access, stating that broad network access is amongst one of five essential characteristics to a cloud system. They establish that at its core, a Cloud system is intended to provide systems that allow access from any standard device over shared resources. Mell and Grance (2011) further discusses the specific services provided by cloud systems into three categories, Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). All three services utilize cloud computing to transport the usage of their functionality to a network connection. The impact of these services is verified by their dominating presence in the most visible applications and tools available within the information technology landscape. Facebook, Gmail, One Drive, and Dropbox are all examples of SaaS that are used on a daily basis by millions of individuals from their phones, tablets, and personal computers, with each access point affecting network traffic. The increase in network traffic grows exponentially when considering Infrastructure as a Service. IaaS is now making it possible for businesses to place their entire enterprise infrastructure into the cloud, creating a scenario where all employees are accessing systems through a network connection.

Cloud computing created the platform for two very big trends in the information technology landscape that contribute heavily to the traffic being generated. The Bring Your Own Device (BYOD) movement is the first trend that has risen parallel to cloud computing technology. With the swarm of services now available from any network access point, businesses have begun to raise employee productivity by provisioning them access to company assets from their own mobile devices. The benefit of each

employee being constantly connected to their email can quickly cause problems for a network. Employees that once were a single user on a network, generating information only when accessing data that was not on their local machines, are now a source for multiple points of access. A cell phone, tablet, and laptop are commonplace and an employee could very easily be carrying all three at the same time. Each device, even when not in direct use, may be attempting to access the network to receive updates or patches. Applications on these devices could be running, causing congesting to the network even further. In a discussion on how Big Data techniques can help in the BYOD arena, Thor Olavsrud (2012) describes the impact BYOD had at the University of Texas. The 350-acre campus could have at any one time up to 120,000 individual devices connected to the network with the users representing the devices numbering in the tens of thousands. This gap between users and devices will continue to grow as the second trend Cloud Computing helped put on the map evolves. The Internet of Things (IoT) describes the multitude of smart devices that communicate through a network connection. IoT devices can be found in a massive amount of industries and systems, such as smart lighting, thermostats, and other home safety devices. While making the everyday lives of individuals more convenient, the impact that these devices have on network traffic is dangerously high. Analysts at Gartner (2015) predict that by during the year 2016 6.4 billion IoT devices will be connected to networks worldwide.

Table 1

Internet of Things Install Projections

Internet of Things Units Installed Base by Category (Millions of Units)

Category	2014	2015	2016	2020
Consumer	2,277	3,023	4,024	13,509
Business: Cross-Industry	632	815	1,092	4,408
Business: Vertical-Specific	898	1,065	1,276	2,880
Grand Total	**3,807**	**4,902**	**6,392**	**20,797**

Source: Gartner, 2015

It is evident that network traffic will continue to grow as more devices are moved to the cloud. This influx of data is potentially crippling for Intrusion Detection Systems (IDS). An IDS must maintain reliable, precise and complete information in relation to the system they are protecting (Pranggono, Mclaughlin, Yang, & Sezer, 2013). If IDSs do not have the capability to be able to view all of the data within their scope, there is no way to ensure all the possible threats have been viewed. Eugene Albin explores this within his comparison of Snort and Suricata intrusion detection systems. Albin (2011) found that both tools had bandwidth issues that directly affected each IDSs' ability to monitor live traffic over a 20Gbps network. The limitations of Intrusion Detection Systems are also explained by Weirong Jiang and Viktor K. Prasanna in the context of their use in discovering signature based attacks. A signature-based attack discovery relies on a set of rules that the IDS can use to detect an intrusion pattern (Scarfone & Mell, 2007). Jiang and Prasanna (2013) discuss that the signature-based detection utilized today has a bottleneck effect in networks with heavy traffic in which Ethernet link rates are pushed beyond the 100Gbps limits.

Literature Related to the Problem

The volume of network traffic being generated by modern systems has quickly been classified under the grouping of Big Data. It is within this field that many innovations were made which provided solutions to only a small number of data analysis problems. Big Data's core components consist of the three v's, volume, velocity, and variety (El Jamiy, Daif, Azouazi, & Marzak, 2014). The volume focuses on the size of data, velocity describes the speed that the data is created, and the variety of the data relates to the unstructured nature. In El Jamiy et al. (2014) Big Data framework, the team evaluates the challenges created by Big Data issues and focuses on data analysis for a portion of the paper. The team claims current tools have become outdated when examining unstructured data. In a Survey of Log Analysis and management, Thosar, Mane, Raykar, Jain, Khude, and Guru (2015) directly address this problem by providing several tools and techniques to combat these issues through Big Data Analytics such as Hadoop and MapReduce frameworks. Both of these technologies stemmed from research started at Google in order to solve their own big data issues. The Hadoop Distributed File System (HDFS) is the storage component of the two and creates a Master/Slave node framework that allows for multiple process requests to be spread from the master across multiple slave nodes (Holmes 2012). The system is optimized for high throughput and is configured to work best with large files of at least the gigabyte range. HDFS provides data replication and fault tolerance as well, duplicating files across multiple nodes based upon software configuration.

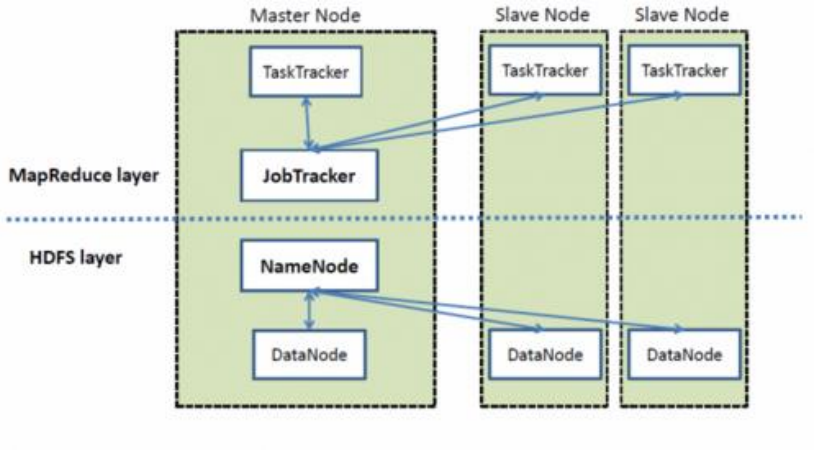


Figure 1. High-Level Hadoop Architecture from “An introduction to Apache Hadoop,” 2014

MapReduce is a programming technique that works on key pairs within two phases, map and reduce. Pioneered by Jeffery Dean and Sanjay Ghemawat, (2004) the two phases run sequentially with the Map phase output becoming the input of the Reduce phase. The process works on determining key pairs, with the Map phase discovering all of the value pairs and the reduce phase placing similar keys together to determine the final output.

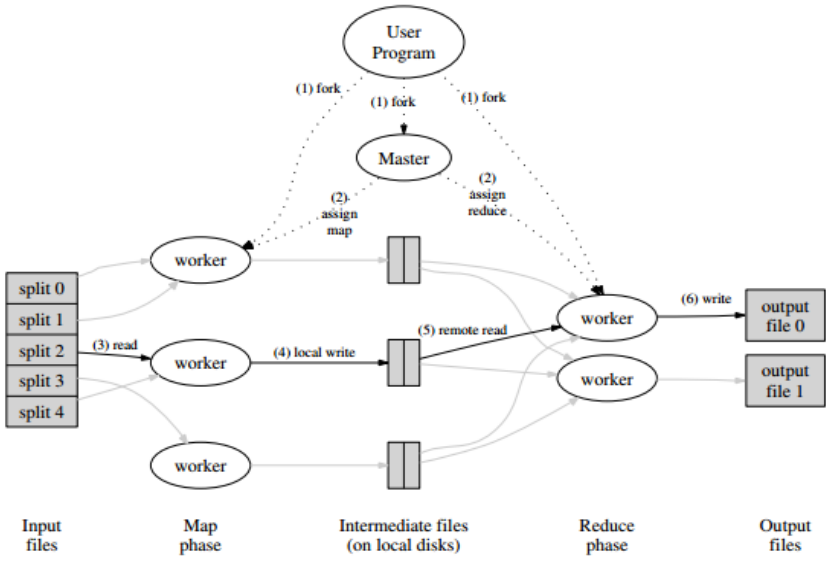


Figure 2. MapReduce Diagram from Dean & Ghemawat, 2004

This process provides a way for data sets that fall into the Big Data category to be analyzed efficiently. The parallel nature of the MapReduce code allows the amount of Hadoop nodes to be scaled up or down based upon the resource need of the system.

Progress was made in processing large volumes of log information, when using Hadoop, MapReduce, and other Big Data techniques. For example, Jeong Jin Cheon and Tae-Young Choe proposed utilizing Hadoop to process logs from the well-known open source IDS Snort. Cheon and Choe (2013) established an eight node Hadoop cluster that processed Snort log information 4.2 times faster than a single system. The distributed system was found to lack real-time scalability, as a result of Snort software limitations. A large number of tests followed and numerous options were discovered. The tests ranged from Pig and Hive as SQL based front ends for Hadoop to entirely

new Big Data platforms, such as Apache Spark, which reports providing 100 times the performance of Hadoop in certain scenarios (“Apache Spark,” 2016).

In light of current Intrusion Detection Systems in the previous examples, hardware modifications were looked into as a possible source for speed increase. Several instances of Snort performance being amplified by GPU implementations were discovered. To offset Snorts poor performance with current multi-threaded processor, Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis (2008) attempted to utilize a GPU to perform the pattern matching process. They were able to assign each individual packet to the multiprocessor of the GPU, Snort output to be raised by a factor of 2. While a success, the experiment was proved inadequate as the team needed to confront several complications of the GPU hardware such as dealing with memory buffers, microprocessors, and learning a new programming language in order to work with the GPU itself. The Snort experiment shows the power of a GPU works well within a parallel system, such as MapReduce on Hadoop, but the difficulties working directly with the GPU have prevented this technique from being widely adopted.

Literature Related to the Methodology

After establishing the benefits of both GPU processing and the MapReduce platform, research was done to determine if there was an easier way to take advantage of the parallel processing power of each system. The main difficulty in processing the MapReduce queries resided in the complexity of porting Java code to a GPU

environment. The Hadoop framework is implemented in Java and, like MapReduce often utilizes the language as well. This information, however, led to the creation of Aparapi. Aparapi is a Java API that was designed by AMD to improve Java applications by allowing them to be easily written for GPUs (Joshi, 2012). It allows programmers to only send the code that performs a parallel action to the GPU. This is done by initiating an Aparapi kernel that converts the intended java code to Open CL with no previous Open Cl training required for the programmer (Joshi, 2012). The platform takes the functionality one step further by still allowing the code to run in a CPU based system. This way, the same code can be used to run jobs on any system regardless of the hardware available within the system. In a discussion on the importance of outliers in Intrusion Detection, Ahmed, Mahmood, and Hu (2013) discussed that modern outlier detection techniques need to be computationally effective in order to handle the large amounts of data that systems are now faced with. Due to limitations within the Aparapi code base, it currently does not work in conjunction with many of the primitives necessary to allow easy porting of native Java code, most notably the String type. The code was created throughout this process, keeping possible future improvements to the API that will allow for native Java containing such values to be easily ported to a GPU-based system.

Summary

The previous chapter provided an understanding of the problems faced by the amount of network data generated from cloud computing and the services the

technology supports. As Intrusion Detection Systems have begun to suffer under the weight of this data, new tools such as Hadoop, MapReduce, custom Java coding and GPU processing have been brought to task to create new ways to analyze network data. The following chapter will provide a multistep platform combining the previously discussed hardware and software tools and will lay out a methodology for determining at which point these systems will be needed. Custom java programs will be utilized in the hopes that the Aparapi functionality improves in the future and that they can be easily ported to a GPU-based system.

Chapter III

METHODOLOGY

Introduction

In chapter three, a study design of the experiment will be discussed and a high-level overview will be provided. This includes a detailed account of the pieces that encompass the experiment as a whole such as MapReduce, Hadoop Clustering, and Amazon Web Services. In addition, the process of data collection is explored, offering insight on how and with which tools, the results were determined, collected, stored, and analyzed.

Design of the Study

A quantitative study design was completed that collected the CPU execution time of MapReduce programs across a 4 node Hadoop cluster and native Java programs across a single node. This form of research was done to obtain a one to one comparison of each processor run time in relation to the code being executed on various sizes of network traffic log information. The design of the study will be broken down into the following categories: Sample Data, Data Analysis Goals, MapReduce Code, Java Code, and Hardware Setup.

Sample Data

The data to be analyzed was obtained from the edge node of the St. Cloud State University. Data was retrieved in PCAP format that, due to complications with the Hadoop-Pcap library, was further manipulated using Tshark to pull specific fields from each individual packet capture. The following fields were obtained and sent to an output text file.

Table 2

Tshark Variables for PCAP Manipulation

Tshark field variable	Description
frame.time	Timestamp of packet capture
ip.proto	Specific IP Protocol of the packet
ip.src	Source IP Address
tcp.srcport	Source Port
ip.dst	Destination IP address
tcp.dstport	Destination Port
ip.ttl	Time To Live value
ip.len	Total Packet Length
ip.flags	IP Flags sent in hex value

This capture was performed by the following command line syntax:

```
tshark -T fields -n -r 20160822-1818-1828.cap -e frame.time -e ip.proto -e ip.src -e
tcp.srcport -e ip.dst -e tcp.dstport -e ip.ttl -e ip.len -e ip.flags > data.txt
```

This provides a uniform data sample that can be run easily through MapReduce jobs.

```
Aug 22, 2016 18:27:57.637652000 CDT     6       199.17.18.73    48407   216.58.217.102  443     63      52      0x00000002
Aug 22, 2016 18:27:57.637674000 CDT     6       199.17.18.73    48407   216.58.217.102  443     63      52      0x00000002
Aug 22, 2016 18:27:57.637695000 CDT     6       199.17.18.73    48407   216.58.217.102  443     63      52      0x00000002
Aug 22, 2016 18:27:57.637736000 CDT     6       199.17.18.73    48407   216.58.217.102  443     63      98      0x00000002
```

Figure 3. Sample Input Data

The final step in the source data manipulation was to create multiple size variations of the data. This was done with a simple script to append original file text content twice to a new file. This step created a new file that was twice the size of the previous. The initial pcap data provided was 20 MBs and this process was repeated generating test sets all with a maximum size of 2.6 GBs.

```
cat 20MB_input.txt 20MB_input.txt > 40MB_input.txt
cat 40MB_input.txt 40MB_input.txt > 80MB_input.txt
cat 80MB_input.txt 80MB_input.txt > 160MB_input.txt
cat 160MB_input.txt 160MB_input.txt > 320MB_input.txt
cat 320MB_input.txt 320MB_input.txt > 640MB_input.txt
cat 640MB_input.txt 640MB_input.txt > 1.3GB_input.txt
cat 1.3GB_input.txt 1.3GB_input.txt > 2.6GB_input.txt
cat 2.6GB_input.txt 2.6GB_input.txt > 5GB_input.txt
```

Figure 4. Scripting for Data Set Creation

Table 3

Data Set File Size and Number of Records

File Size (MB)	File Size in Bytes	Number of Records in file
19.9800005	20950549	224874
39.96000099	41901098	449748
79.92000198	83802196	899496
159.840004	167604392	1798992
319.6800079	335208784	3597984
639.3600159	670417568	7195968
1278.720032	1340835136	14391936
2557.440063	2681670272	28783872
5114.881260	5363340544	57567744

Upon creation, all data was stored on its respective EC2 instance and was placed within the Hadoop File Structure (HDFS) of the Hadoop cluster used for testing.

Data Analysis Goals

In order to test the efficiency of the two systems, four specific analytical outputs were determined based upon the fields that were pulled from the pcap data: TCP Port usage by Source IP, Total Traffic by Source IP, Average Packet Length by Source IP, and Total Traffic by IP Protocol. Each output provided a different challenge in terms of processing the output that MapReduce and Java handled differently. The initial selection of this output was determined by the factors notated below.

Table 4

Job Output Characteristics

Desired Output	**Challenge in acquiring**
TCP Port usage by Source IP	High volume of unique keys
Total Traffic by IP Protocol	Low number of unique keys
Average Packet Length by Source IP	Mathematics with small variable values
Total Traffic By Source IP	Mathematics with large variable values

MapReduce Code

The MapReduce code for each program written can be broken down into two main sections; Mapping and Reducing. During the Mapping phase, a specific key value was obtained for each program from the input data and matched with an output value that was used during the reduce phase. For the Map input, the tshark manipulated data

is tab delimited, so as each individual line is read in by the Map process, it was split into a temp array of strings.

Ex. Input Split - String temp[] = value.toString().split("\t");

Table 5

Temp Array Relation to Tshark Input Data

Array Value	Tshark Input Data
Temp[0]	Timestamp
Temp[1]	IP Protocol
Temp[2]	Address Source IP
Temp[3]	Source Port
Temp[4]	Destination IP Address
Temp[5]	Destination Port
Temp[6]	Time To Lve (TTL)
Temp[7]	Packet Length
Temp[8]	IP Flags

Splitting each entry into an array provided the ability to easily select the value to be used as the key value as well as access any additional packet information during the mapping function. Once all the lines of the input have been processed, the values for all related to the unique keys are summed. Each of the MapReduce programs used a slightly different process to provide the intended results, with some simply counting unique keys, and others associating the key to a value from the input stream.

MapReduce Program 1 - Volume of TCP Ports by Source IP

To map the Source IP and the Port together, the following mapping was utilized to first determine if the packet data was sent using TCP. If TCP was used for this packet, a unique key was created by creating a string combining the Source IP and the

Source Port. This Key was associated with a static 1 value. During the reduce phase, each instance of the unique keys is counted, by summing the related static 1 value.

```
if (temp[1].contains("6"))
    out_map = temp[2] + " " + temp[3];  //Looks for only protocol 6 data and sets each mapped value to SourceIP Port
context.write(new Text(out_map),new IntWritable(1)); //Sets integer at 1 value for each output of the Source Port combination
```

Figure 5. Map Function of TCP Port by Source IP

```
int sum = 0;
for (IntWritable val : values)
{
    sum += val.get(); //stores total number of each individual key incrementing off of the mapped integer of 1
}
context.write(key, new IntWritable(sum));
```

Figure 6. Reduce Function of TCP Port by Source IP

Map Reduce Program 2 - Total Traffic by IP Protocol

The total amount of traffic for each protocol was determined by using the IP protocol number as the Key value and is mapped to the total length of the packet. With the total length of each packet as the counter value to be summed for each mapped instance, this provided a simple way to determine the total traffic for each protocol. Also, during the reduce phase, basic mathematical manipulation is performed to convert the packet length from bytes to megabytes in order to reduce the end value size.

```
    output.set(Float.parseFloat(temp[7])); //Sets the output value to the packet length of the input line as an integer
    context.write(new Text(temp[1]),output);      //Sets a mapping of the protocol number to the the integer value of the packet length
```

Figure 7. Map Function Total Traffic by IP Protocol

```
float sum = 0;
for (FloatWritable val : values)
{
    sum += (val.get()/1024)/1024;  //sums the total traffic in bytes for each unique protocol number
}
context.write(key, new FloatWritable(sum));  //Writes output of protocol number and total traffic
```

Figure 8. Reduce Function Total Traffic by IP Protocol

MapReduce Program 3 - Average Packet Length by IP Source

Average Packet Length mapped data in the same manner as the IP protocol code, substituting the Source IP for the Protocol in order to obtain the total amount of traffic for each IP address. The Reduce function includes a counter that increments as each instance of a key is processes. This count variable determined the total number of packets and was then used to obtain the average packet length from the total packet traffic per that specific IP address.

```
output.set(Float.parseFloat(temp[7]));        //Sets the output value of the map to the packet length
context.write(new Text(temp[2]),output);      //Outputs a mapping of the source IP and the packet length
```

Figure 9. Map Function Average Packet Length by Source IP

```
float sum = 0;   // value to hold total traffic for each unique IP
float avg = 0;   // variable to hold average value
float count = 0; // count variable to determine how many packets for each unique IP
for (FloatWritable val : values)
{
    sum += val.get();      //sums the value of each unique IP
            count++;;      //increments counter to be used in average calculation
}

avg = sum/count;           //calculates average
context.write(key, new FloatWritable(avg)); //provides output mapping
```

Figure10. Reduce Function Average Packet Length by Source IP

Map Reduce Program 4 - Total Percentage of Traffic by IP Source

Data manipulation was performed during the Map phase instead of the Reduce phase in order to obtain the total traffic percentage for each IP address. The packet length is mapped to each source IP address, but prior to mapping is converted into megabytes. This conversion is done in order to dramatically lower the variable footprint of the total traffic static variable. During the mapping of each input line, the packet length is added to this static value in order to provide an overall total traffic value used to calculate the percentage during the reduction phase. The reduction performed basic mathematical functions to determine the overall percentage based upon the total traffic and the reduced sum for each mapped IP address.

```
String temp[] = value.toString().split("\t"); //Reads each line of input file into an array split by tabs
float val_temp = Float.parseFloat(temp[7]);
val_temp = (val_temp/1024)/1024;

Total_traffic+=val_temp;     //increments the total traffic value
context.write(new Text(temp[2]),new FloatWritable(val_temp));      //maps each source iP to a traffic value
```

Figure 11. Map Function Percent of Traffic by Source IP

```
float sum = 0;

for (FloatWritable val : values)
{
    sum += val.get();   //sums total traffic for each unique IP
}

context.write(key, new FloatWritable(((sum/Total_traffic)*100))); //returns string format in order to limit number of decimals
```

Figure 12. Reduce Function Percent of Traffic by Source IP

Java Code

In order to mimic the MapReduce functionality as much as possible, a similar thought process was used to create the single node Java code. The idea of unique keys was ported to the Java code in the form of a unique hash set. The hash set was used to determine the unique values that would be used for mapping a value in order to determine the desired output. For each program, a buffered file reader was used to read in the data from the text input file. As data was read in, the same temp array process as the MapReduce job was used to read each line. It is at this point the same key value from the MapReduce job was used to assign the unique values to a hash set. The Hash set allows each line to be analyzed, and if the value from the temp array chosen is not within the set, it is added in. If the value has already been added, there is no action. This hash set was then converted to an array of strings with any number of corresponding arrays to store other key values such as counter values or sum values. The input file was then scanned again for each individual unique value, and for each instance of the unique key, an action is performed on the corresponding arrays. The final step is to produce the desired output from the multiple arrays that have been used to collect all necessary data. Each individual program's unique hash and data collection are discussed below.

Program 1 - Volume of TCP Source Ports by IP

The same Source IP and Source port are used in the Java program as in the MapReduce to determine the unique keys. In addition to the unique IP address array, a

counter array was established to keep count of how many instances of each IP/Port combination were discovered within the input file. The output is generated by printing the output of the unique array in conjunction with the total count of each instance collected during the multiple file read process.

```
while ((line = reader.readLine()) !=null)
{
    String temp[] = line.toString().split("\t"); //Reads each line of input file and splits it by tab into a temp array
    uniqueLine.add(temp[2]+ " " + temp[3]); //adds IP source and Port as a unique value to hash set
}
reader.close();
```

Figure 13. Java TCP Port by Source IP Unique Key Value/Assignment

```
String[] uniqueArray = uniqueLine.toArray(new String[uniqueLine.size()]);
int[] countArray = new int[uniqueLine.size()];
```

Figure 14. Java TCP Port by Source IP Java Arrays

```
for(int j=0;j<uniqueArray.length;j++){
    if((temp[2]+ " " + temp[3]).equals(uniqueArray[j])){
         countArray[j]++;
        }
    }
```

Figure 15. Java TCP Ports by Source IP Count of Unique IP/Ports

```
for(int i=0;i<uniqueArray.length;i++)
  bw.write(uniqueArray[i] + " " + countArray[i]  + "\n");
```

Figure 16. Java TCP Ports by Source IP Output Generation

Java Program 2 - Total Traffic per IP Protocol

The same was used from the corresponding MapReduce program. One additional array was created in order to track the total traffic volume for each unique protocol in the uniqueProtocol Array. The total traffic was calculated by adding the packet length of each input line to the array field corresponding to the unique protocol value found within each input line.

```
while ((line = reader.readLine()) !=null)
{
   String temp[] = line.toString().split("\t"); //Reads each line of input file and splits it by tab into atemp array
   uniqueProtocol.add(temp[1]); //Adds only unique protocol values into the Unique HashSet uniqueIPs
   }
   reader.close();
```

Figure 17. Java Total Traffic per IP Protocol Unique Key Value/Assignment

```
String[] ProtocolArray = uniqueProtocol.toArray(new String[uniqueProtocol.size()]);
float[] sumArray = new float[uniqueProtocol.size()];
```

Figure 18. Java Total Traffic per IP Protocol Java Arrays

```
for(int j=0;j<ProtocolArray.length;j++){
    if(temp[1].equals(ProtocolArray[j])){
        k= Float.parseFloat(temp[7]);
        sumArray[j]+=(k/1024)/1024;
        }
    }
```

Figure 19. Java Total Traffic per IP Protocol Sum Traffic per Protocol

```
for(int i=0;i<ProtocolArray.length;i++)
{
 bw.write("Protocol = " + ProtocolArray[i] + " Total = " + sumArray[i] + " megabytes\n");
}
```

Figure 20. Java Total Traffic per IP Protocol Output Generation

Java Program 3 - Average Packet Length by IP Source

The Source IP was used as the unique key in relation to the average packet length. Three arrays are established to contain unique IPs, a sum of traffic for each unique IP, and the total number of packets for each unique IP. The average calculation was performed during the output creation while the file is being written to the buffered writer.

```
while ((line = reader.readLine()) !=null)
{
    String temp[] = line.toString().split("\t");  //Reads each line of input file and splits it by tab into a temp array
    uniqueIPs.add(temp[2]);   //Adds only unique IP addresses into the Unique HashSet uniqueIPs
}
reader.close();
```

Figure 21. Java Average Packet Length Unique Key Value/Assignment

```
String[] IPArray = uniqueIPs.toArray(new String[uniqueIPs.size()]);  //Converts IPs to array of strings
float[] sumArray = new float[uniqueIPs.size()]; //Array that will correspond with IPArray for store total traffic per IP
float[] countArray = new float[uniqueIPs.size()]; //Array that will correspond with IPArray to store total number of packets per IP
```

Figure 22. Java Average Packet Length Java Arrays

```
for(int j=0;j<uniqueIPs.size();j++){
  if(temp[2].equals(IPArray[j])){
      k= Float.parseFloat(temp[7]);
      sumArray[j]+=k;
      countArray[j]++;
      }
  }
```

Figure 23. Java Average Packet Length Traffic Summation and Packet Count

```
for(int i=0;i<uniqueIPs.size();i++)
{
 bw.write(IPArray[i] + "\tTotal =\t" + sumArray[i] + "\tAvg =\t" + (sumArray[i]/countArray[i]) + "\n");
}
```

Figure 24. Java Average Packet Length Output Generation

Java Program 4 – Total Percentage of Traffic by Source IP Address

Source IP was used as the unique key. A sumArray was created in addition to the uniqueIP array to store the total traffic associated with each unique IP address. Each unique IP processed through the input file, and incremented the associated sum array location with the packet length when a match between the IP address temp array field matched the unique IP. Also, when the traffic value is added to the sumArray, it is also added to the totalTrafic variable for percentage calculation.

```
while ((line = reader.readLine()) !=null)
{
     String temp[] = line.toString().split("\t"); //Reads each line of input file and splits it by tab into a temp array
     uniqueIPs.add(temp[2]);  //Adds only unique IP addresses into the Unique HashSet uniqueIPs
}
reader.close();
```

Figure 25. Java Percentage of Traffic by IP Unique Key

```
String[] IPArray = uniqueIPs.toArray(new String[uniqueIPs.size()]); //Converts IPs to array of strings
float[] sumArray = new float[uniqueIPs.size()]; //Array that will correspond with IPArray for store total traffic per IP
```

Figure 26. Java Percentage of Traffic by IP Java Arrays

```
for(int j=0;j<IPArray.length;j++)
{
if(temp[2].equals(IPArray[j]))
{
    sumArray[j]+=(Float.parseFloat(temp[7])/1024)/1024;  //adds to sum of unique IPs traffic
    Total_traffic+=(Float.parseFloat(temp[7])/1024)/1024;   //Adds to total traffic
}
}
```

Figure 27. Java Percentage of Traffic by IP Traffic Calculations

```
for(int i=0;i<IPArray.length;i++)
{
   float percent;
   percent = (((sumArray[i]/Total_traffic)*100));
   if(percent <.001)
   bw.write(IPArray[i] +"\t " +  df.format(percent) + "\n"); //generates output
}
```

Figure 28. Java Percentage of Traffic by IP

Hardware Setup

All systems used during testing and research were established within an Amazon Web Services environment. T2.medium Elastic Cloud Compute (EC2) servers were used as needed in an effort to keep costs to a minimum while processing. This server was chosen as its specs relate the closest to a low-end system that could be used in a production environment today.

Table 6

AWS Hardware Costs

	vCPU	ECU	Memory (GiB)	Instance Storage (GB)	Linux/UNIX Usage	GB per month
General Purpose - Current Generation						
t2.nano	1	Variable	0.5	EBS Only	$0.0065 per Hour	$0.10
t2.micro	1	Variable	1	EBS Only	$0.013 per Hour	$0.10
t2.small	1	Variable	2	EBS Only	$0.026 per Hour	$0.10
t2.medium	2	Variable	4	EBS Only	$0.052 per Hour	$0.10

Source – "EC2 Instance Types,"2016

The Hadoop cluster was established with four t2.medium servers, one acting as a name node which controls the Hadoop cluster, and three data nodes that will take care of the processing of data. Each server has a total of 20GBs of allocated storage space, to allow for the all the testing files to be stored. Java tests were run on a single t2.medium server with the same specs and storage as the Hadoop name node to create as close of a parallel between the two hardware profiles as possible.

Data Collection

Each of the eight programs, four MapReduce and four single node Java, were processed a total of five times for each sized data file. The below sample was used to collect data for each of the five tests for each file size.

Table 7

Test Recording Table for one Program at 19.98 MB Size

	Date	Start Time	CPU Time	File Size
Test 1				19.98
Test 2				19.98
Test 3				19.98
Test 4				19.98
Test 5				19.98
Average				

A different date and start time was used for each test to check for any significant differences in processing at different times due to possible resource sharing of the AWS instances used. All code was initiated from the command line of the server using a predefined input string to determine the input and output of each file.

```
hadoop jar SourcePortMap.jar SourcePortMap /input/5GB_input.txt /output/test3/SourcePortMap_5GB
hadoop jar SourcePortMap.jar SourcePortMap /input/2.6GB_input.txt /output/test3/SourcePortMap_2.6GB
hadoop jar SourcePortMap.jar SourcePortMap /input/1.3GB_input.txt /output/test3/SourcePortMap_1.3GB
hadoop jar SourcePortMap.jar SourcePortMap /input/640MB_input.txt /output/test3/SourcePortMap_640MB
hadoop jar SourcePortMap.jar SourcePortMap /input/320MB_input.txt /output/test3/SourcePortMap_320MB
hadoop jar SourcePortMap.jar SourcePortMap /input/160MB_input.txt /output/test3/SourcePortMap_160MB
hadoop jar SourcePortMap.jar SourcePortMap /input/80MB_input.txt /output/test3/SourcePortMap_80MB
hadoop jar SourcePortMap.jar SourcePortMap /input/40MB_input.txt /output/test3/SourcePortMap_40MB
hadoop jar SourcePortMap.jar SourcePortMap /input/20MB_input.txt /output/test3/SourcePortMap_20MB
```

Figure 29. MapReduce Test Input Command

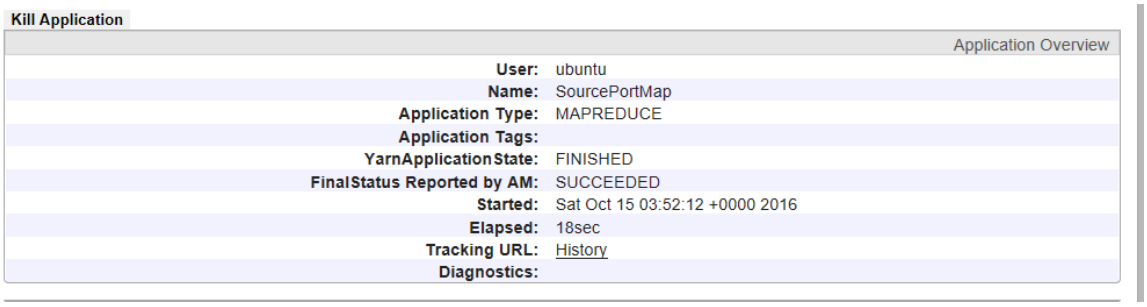
```
java -cp JavaSourcePortMap.jar JavaSourcePortMap ~/Input/5GB_input.txt ~/JavaOut/JavaSourcePortMap_5B.txt
java -cp JavaSourcePortMap.jar JavaSourcePortMap ~/Input/2.6GB_input.txt ~/JavaOut/JavaSourcePortMap_2.6GB.txt
java -cp JavaSourcePortMap.jar JavaSourcePortMap ~/Input/1.3GB_input.txt ~/JavaOut/JavaSourcePortMap_1.3GB.txt
java -cp JavaSourcePortMap.jar JavaSourcePortMap ~/Input/640MB_input.txt ~/JavaOut/JavaSourcePortMap_640MB.txt
java -cp JavaSourcePortMap.jar JavaSourcePortMap ~/Input/320MB_input.txt ~/JavaOut/JavaSourcePortMap_320MB.txt
java -cp JavaSourcePortMap.jar JavaSourcePortMap ~/Input/160MB_input.txt ~/JavaOut/JavaSourcePortMap_160MB.txt
java -cp JavaSourcePortMap.jar JavaSourcePortMap ~/Input/80MB_input.txt ~/JavaOut/JavaSourcePortMap_80MB.txt
java -cp JavaSourcePortMap.jar JavaSourcePortMap ~/Input/40MB_input.txt ~/JavaOut/JavaSourcePortMap_40MB.txt
java -cp JavaSourcePortMap.jar JavaSourcePortMap ~/Input/20MB_input.txt ~/JavaOut/JavaSourcePortMap_20MB.txt
```

Figure 30. Java Test Input Command

For each consecutive test run, MapReduce Output path was modified so that test data could be preserved. To preserve the Java data, the output of each run was moved to a new directory named to represent the day of the test.

Tools and Techniques

For the MapReduce processes, execution times will be collected through MapReduce log information that is generated by the Hadoop resource manager Yarn as the jobs are processed. The output of each job run within the system can be viewed by accessing the public DNS of the namenode on port 8088. The output will provide an elapsed time for the MapReduce job.


Kill Application

Application Overview

User:	ubuntu
Name:	SourcePortMap
Application Type:	MAPREDUCE
Application Tags:	
YarnApplicationState:	FINISHED
FinalStatus Reported by AM:	SUCCEEDED
Started:	Sat Oct 15 03:52:12 +0000 2016
Elapsed:	18sec
Tracking URL:	History
Diagnostics:	

Figure 31. MapReduce Output HTML

Output for the java based processes will be determined by recording functionality placed within the code to determine run times that can be compared to the runtime. The System.currentTimeMillis system variable will be used to determine a start time and end time of the program. The difference of these two will be calculated and reported as output to record the total execution time.

```
long StartTime = System.currentTimeMillis();
long StopTime;
long elapsedTime;


StopTime = System.currentTimeMillis();
elapsedTime = StopTime - StartTime;   //calculates runtime of program
```

Figure 32. Java Time Recording Procedure

All data is to be stored within Microsoft Excel where basic mathematical functions will be used to determine the following for each Java and MapReduce program.

Table 8

Final Analysis Categories

Process Analysis
Average Run time
Data processed in megabytes per second
Data processed in megabytes per Hour
Megabyte per hour cost

Summary

This chapter has provided an extensive overview of the design of the experiment to be performed. The steps taken to obtain the necessary data to evaluate a clustered MapReduce job against the performance of a single node Java program were clearly discussed. The inner workings of the individual Java and MapReduce programs have been presented to allow others to repeat the process as is, or with modifications deemed necessary. In the next chapter the information gathered by the above

methodology will be analyzed to determine how it relates to the hypotheses presented within this paper.

Chapter IV

DATA PRESENTATION AND ANALYSIS

Introduction

An abundant amount of data was collected from 360 separate program tests. The information was condensed to specific charts, tables, and graphs to provide an outline of how this information affects the costs and efficiencies of both MapReduce and Java programs.

Data Presentation

File statistics

Nine data files were used for testing each Java and MapReduce program. Each data set doubled the previous files data size and record count.

Table 9

Input File Statistics

File Name	Number of Records in file	File Size (MB)	File Size in Bytes
20MB_input.txt	224,874	19.98	20,950,549
40MB_input.txt	449,748	39.96	41,901,098
80MB_input.txt	899,496	79.92	83,802,196
160MB_input.txt	1,798,992	159.84	167,604,392
320MB_input.txt	3,597,984	319.68	335,208,784
640MB_input.txt	7,195,968	639.36	670,417,568
1.3GB_input.txt	14,391,936	1278.72	1,340,835,136
2.6GB_input.txt	28,783,872	2557.44	2,681,670,272
5.1GB_input.txt	57,567,744	5114.88	5,363,340,544

Full Test Data Collection

Each MapReduce and Java program was tested a total of five times for the previously discussed source files for a total of forty-five tests per program. Tests were performed on different days and times to check for inconsistencies in run times due to possible resource sharing limitations with Amazon Web Services. A full listing of each program's test runs can be found in Appendix I through Appendix P.

Average Run Times

The average runtime of each MapReduce and Java program was calculated from the five tests performed on each size input file. The TCP Port by IP Source Processes have been removed from the overall comparison and the stand-alone Java comparison, into their own charts as the extended runtime of the Java version of this code skews the chart range. The runtimes of the MapReduce and Java programs are also presented

separate to show how they compare to each other based on the different calculations being performed by each process.

Table 10

All Programs Average Run Times in Seconds

	MR TCP Port by Source IP	MR Traffic by IP Protocol	MR Average Packet Length	MR Percent of Traffic	Java TCP Port by Source	Java Traffic by IP Protocol	Java Average Packet length	Java Percent of Traffic
20 MB	17.20	16.00	16.40	16.20	23.00	0.52	0.93	1.02
40 MB	18.20	17.40	17.00	17.60	45.46	0.87	1.59	1.74
80 MB	18.80	18.20	19.40	19.20	90.64	1.45	3.20	3.24
160 MB	21.40	20.60	21.80	21.60	180.45	2.61	6.16	6.24
320 MB	28.00	26.60	27.80	27.60	361.07	5.06	12.28	13.61
640MB	48.40	42.20	44.00	49.40	723.74	7.96	23.49	27.94
1.3GB	66.60	59.60	67.20	62.40	1449.12	18.81	47.94	55.82
2.6 GB	107.80	73.80	87.60	85.80	2918.01	83.54	112.72	116.12
5.1 GB	206.20	127.00	144.20	146.60	9757.80	166.73	219.29	231.32

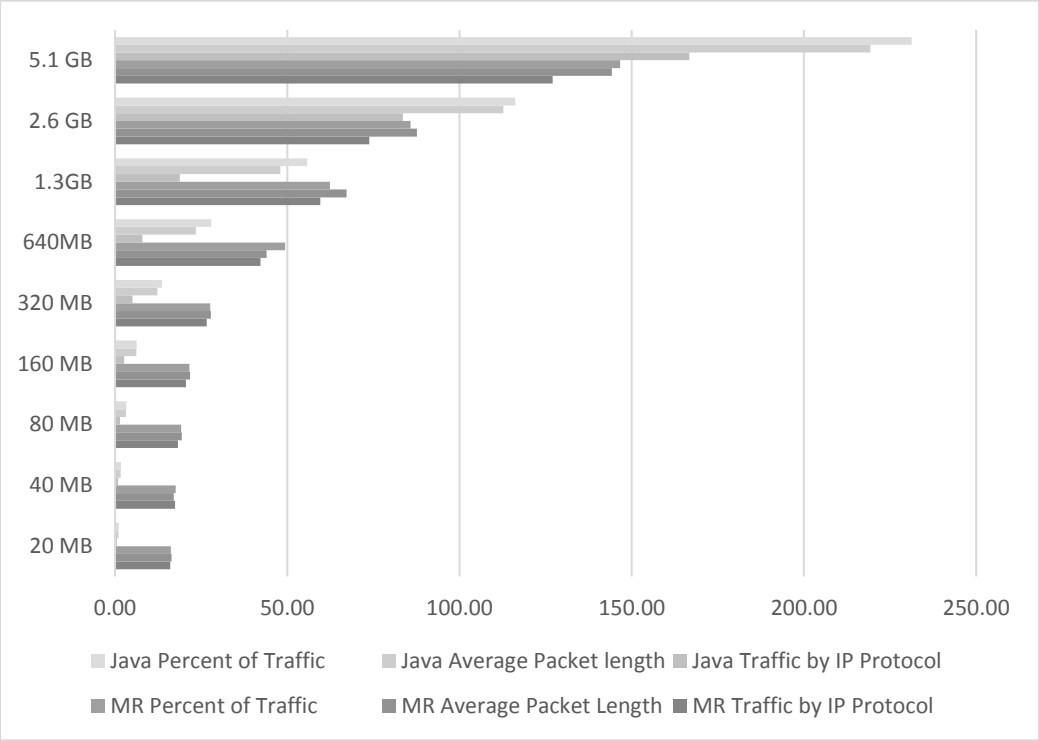


Figure 33. Average Run Time Comparison TCP Port by IP Run Time Removed

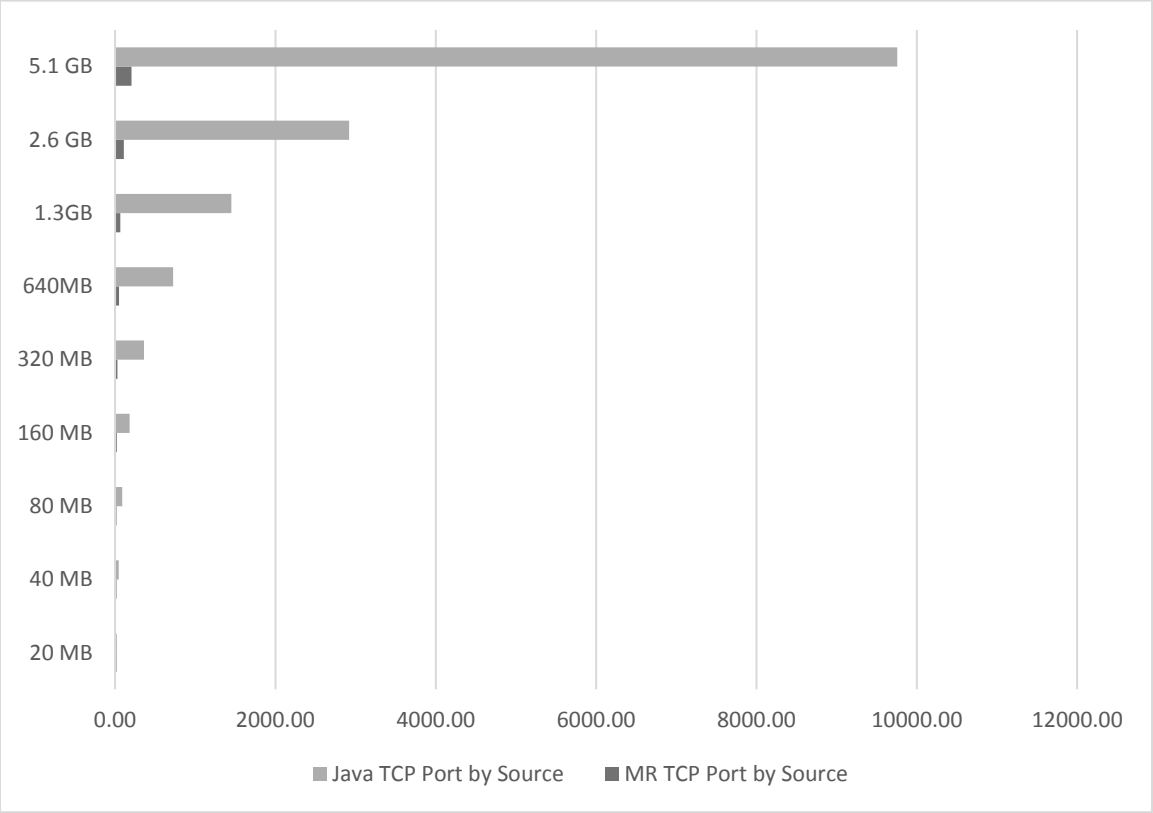


Figure 34. Average Run Time Comparison TCP Port by IP Run Time Only

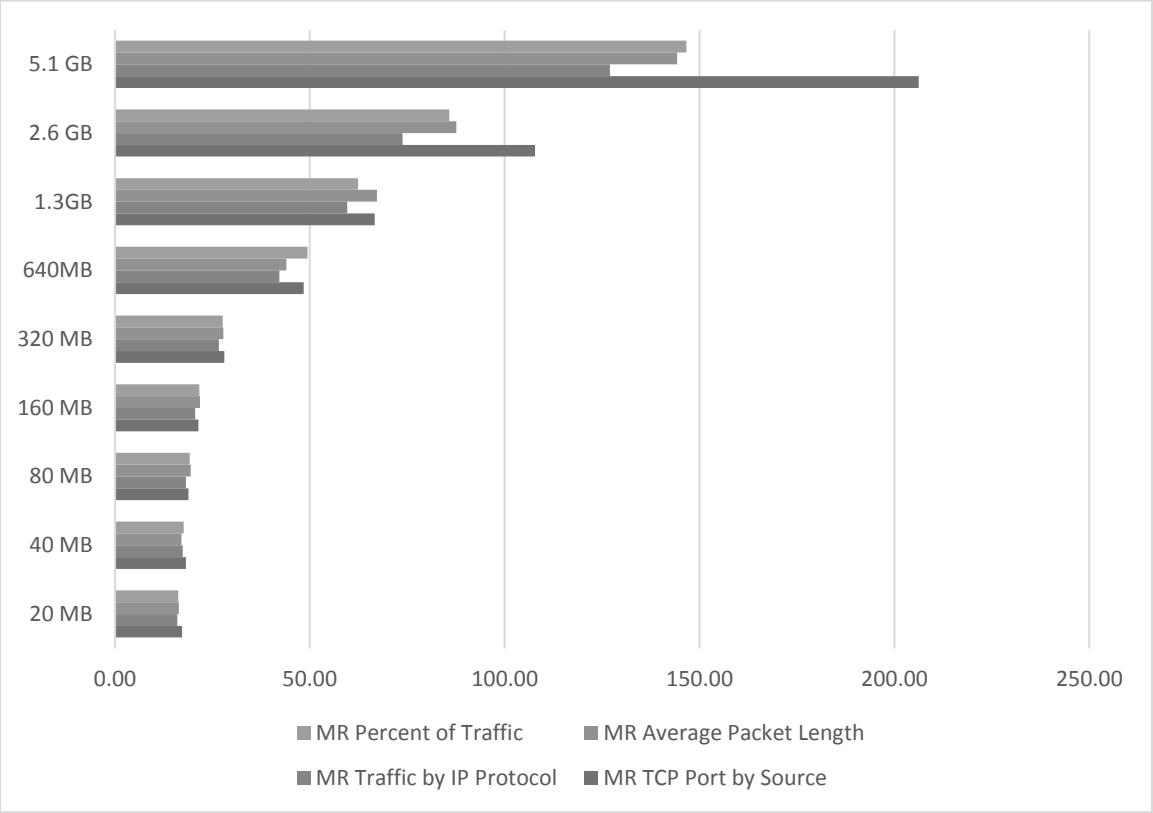


Figure 35. MapReduce Programs Average Run Time in Seconds

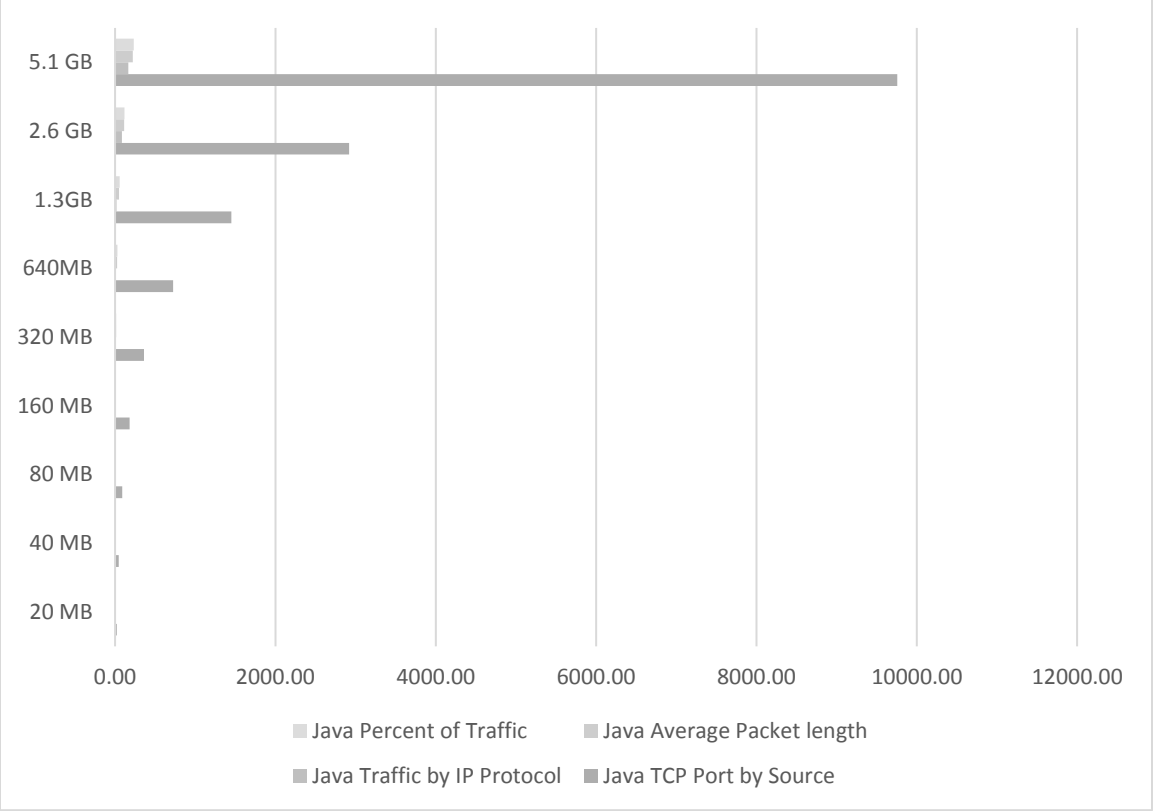


Figure 36. Java Average Run Time in Seconds

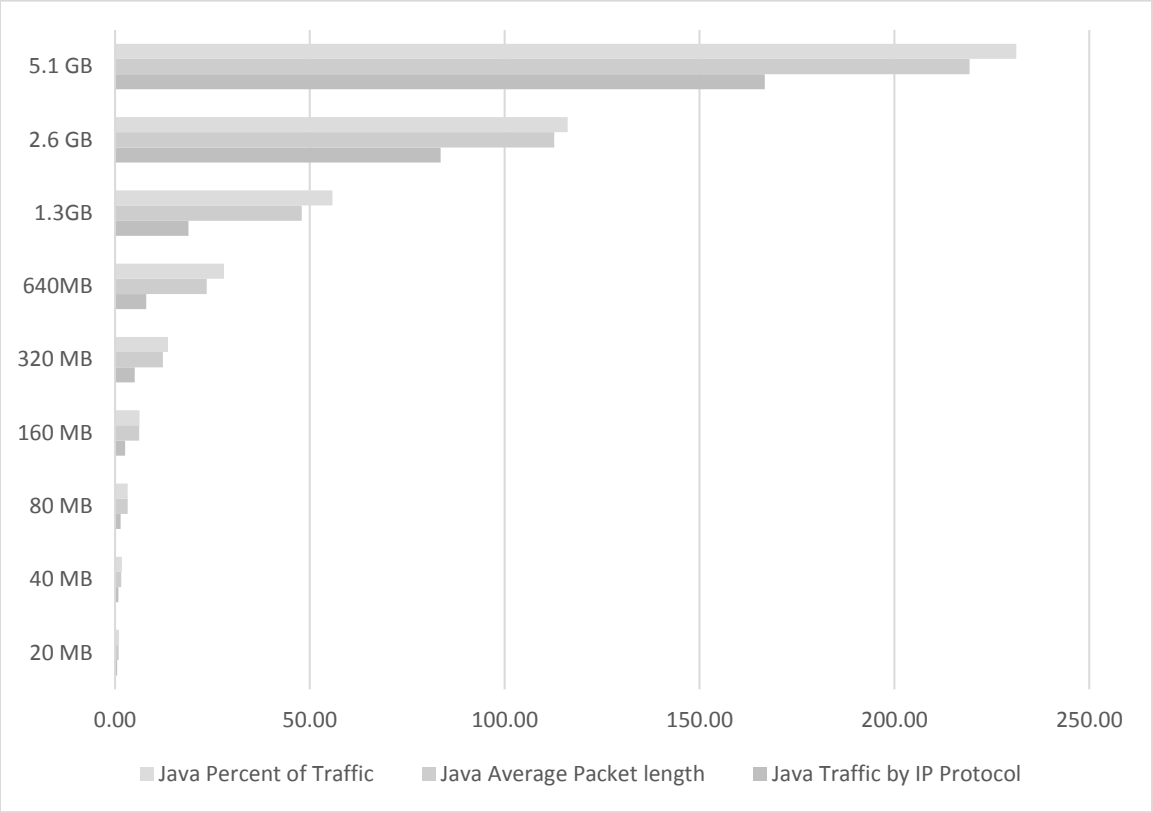


Figure 37. Java Average Run Time in Seconds TCP Port by IP Source Removed

MB per Second Processing time

The overall throughput of each program was affected by the file size that was processed. This information was recorded to determine the most efficient file size for each MapReduce and Java program. To compare the efficiency of each process, each desired output was presented separately to clearly state which program processed data more efficiently.

Table 11

MB/sec Throughput Rate

	MR TCP Port by Source	MR Traffic by IP Protocol	MR Average Packet Length	MR Percent of Traffic	Java TCP Port by Source	Java Traffic by IP Protocol	Java Average Packet length	Java Percent of Traffic
20 MB	1.16	1.25	1.22	1.23	0.87	38.18	21.46	19.62
40 MB	2.20	2.30	2.35	2.27	0.88	45.96	25.08	22.97
80 MB	4.25	4.39	4.12	4.16	0.88	55.17	24.99	24.64
160 MB	7.47	7.76	7.33	7.40	0.89	61.27	25.94	25.61
320 MB	11.42	12.02	11.50	11.58	0.89	63.14	26.04	23.48
640MB	13.21	15.15	14.53	12.94	0.88	80.34	27.22	22.88
1.3GB	19.20	21.46	19.03	20.49	0.88	67.98	26.67	22.91
2.6 GB	23.72	34.65	29.19	29.81	0.88	30.61	22.69	22.02
5.1 GB	24.81	40.27	35.47	34.89	0.52	30.68	23.32	22.11

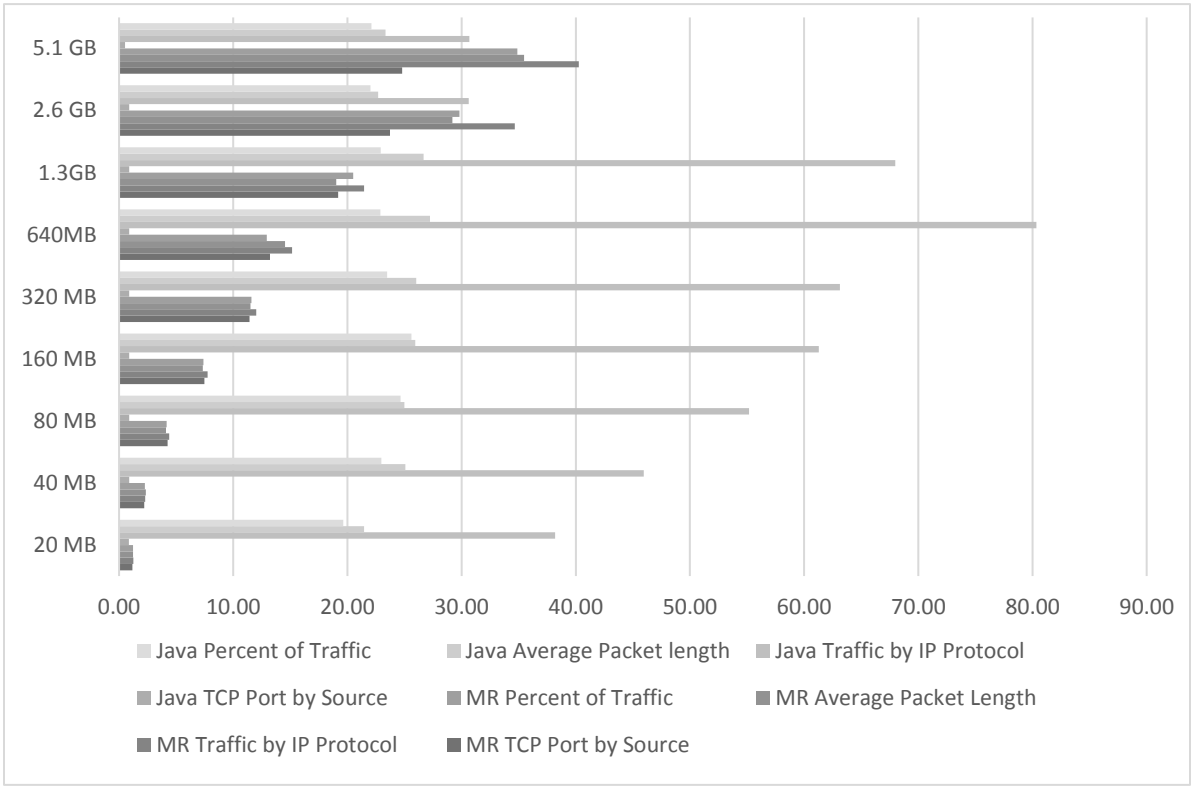


Figure 38. MB/Sec by Input File Size

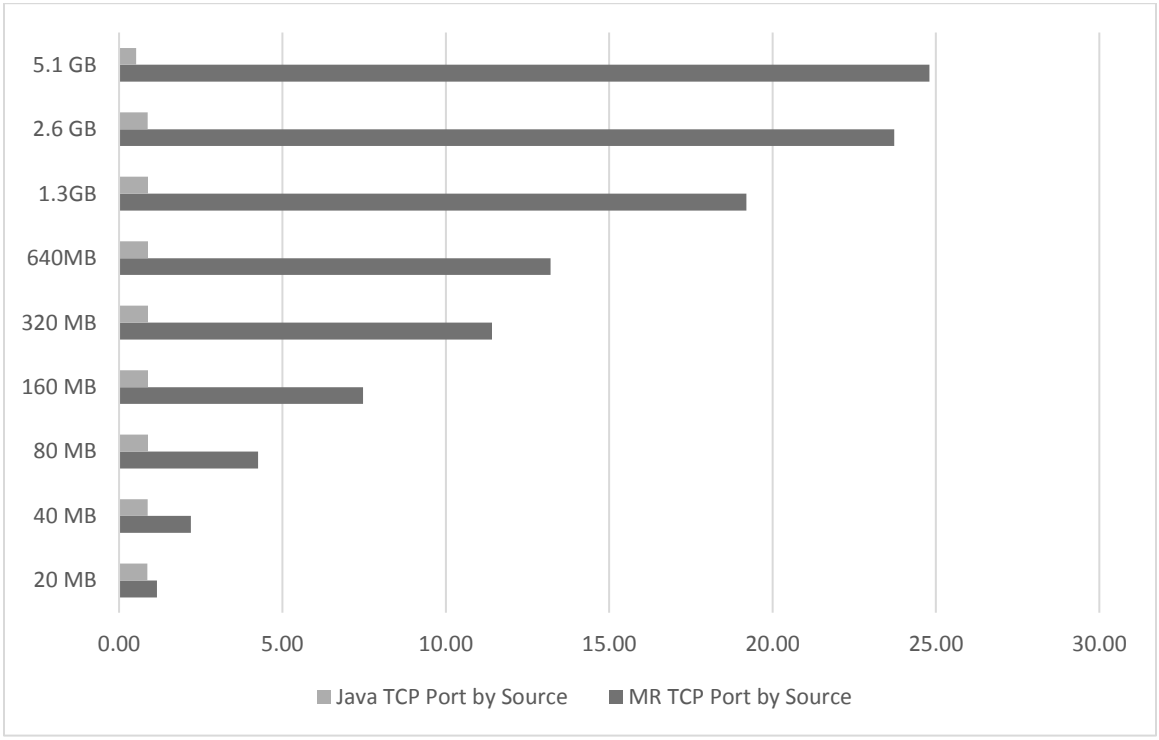


Figure 39. MB/Sec by Input File Size TCP Port by Source IP

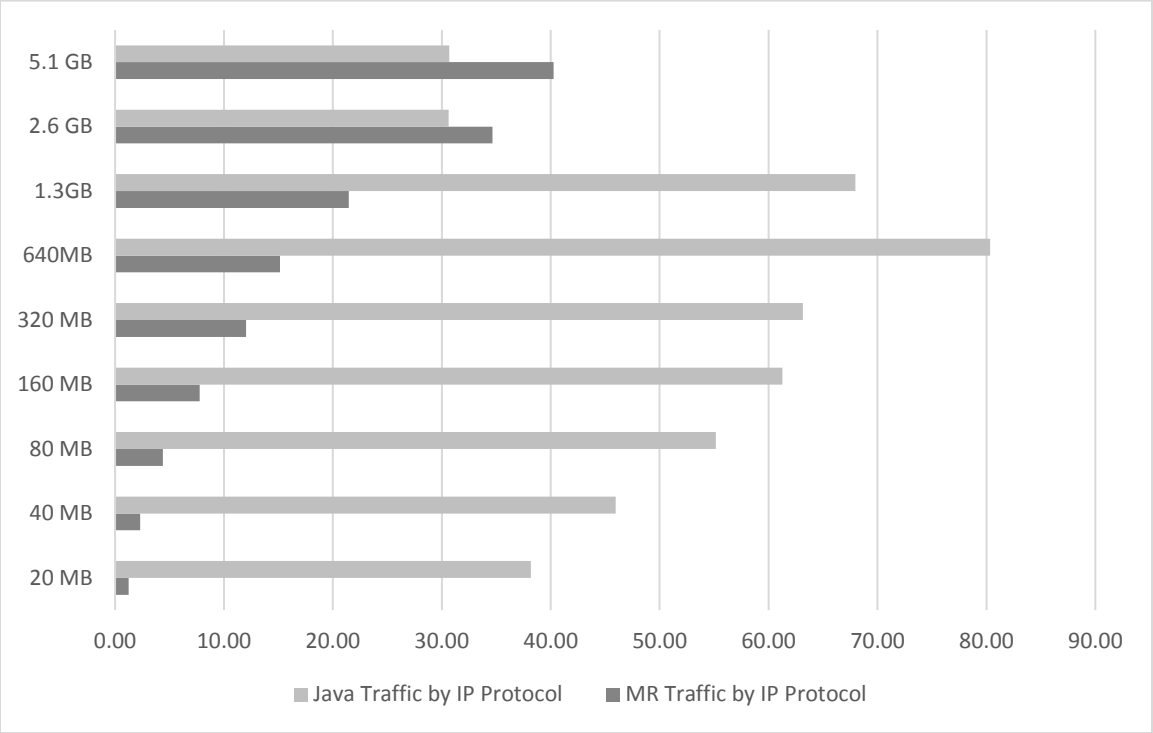


Figure 40. MB/Sec by Input File Size Total Traffic by IP Protocol

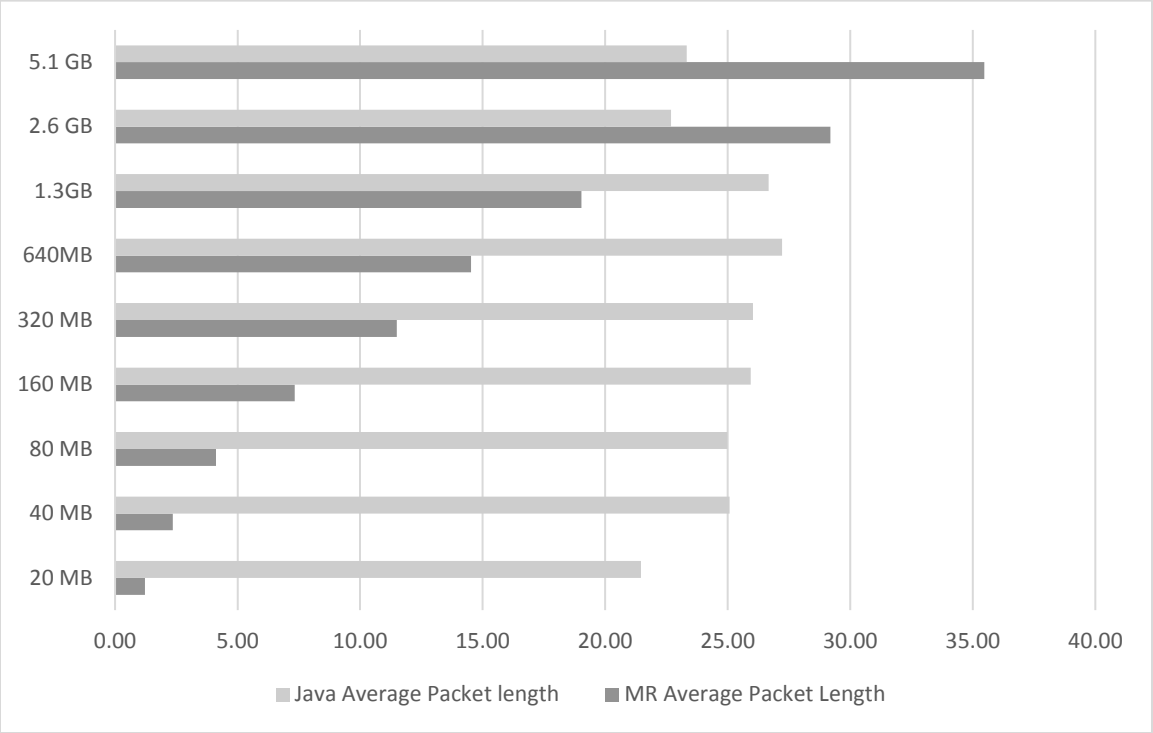


Figure 41. MB/Sec by Input File Size Average Packet Length by IP

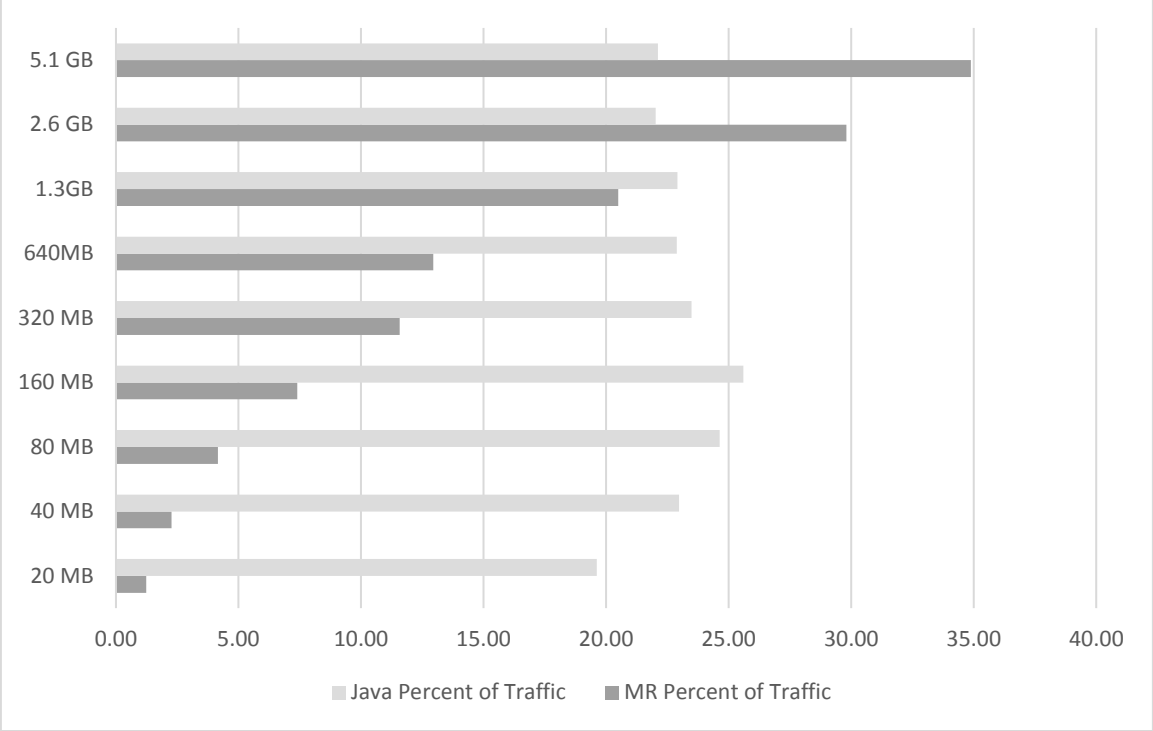


Figure 42. MB/Sec by Input File Size Total Percentage of Traffic by IP

Time to Process (50 Terabytes)

To identify most cost and time efficient scenario a total data value of fifty Terabytes was used when determining the below results. The previous MB/second calculation for each process was used to calculate how long each program would take to provide an output based upon 50 Terabytes of each file size. This overall time was reported in days.

Table 12

Time to Process 50 TB in Days

	MR TCP Port by Source	MR Traffic by IP Protocol	MR Average Packet Length	MR Percent of Traffic	Java TCP Port by Source	Java Traffic by IP Protocol	Java Average Packet length	Java Percent of Traffic
20 MB	522.38	485.94	498.09	492.01	698.57	15.89	28.28	30.92
40 MB	276.38	264.23	258.15	267.27	690.30	13.20	24.20	26.42
80 MB	142.74	138.19	147.30	145.78	688.20	11.00	24.29	24.63
160 MB	81.24	78.21	82.76	82.00	685.07	9.90	23.39	23.70
320 MB	53.15	50.49	52.77	52.39	685.39	9.61	23.31	25.84
640MB	45.94	40.05	41.76	46.89	686.90	7.55	22.29	26.52
1.3GB	31.60	28.28	31.89	29.61	687.68	8.93	22.75	26.49
2.6 GB	25.58	17.51	20.79	20.36	692.37	19.82	26.74	27.55
5.1 GB	24.46	15.07	17.11	17.39	1,157.64	19.78	26.02	27.44

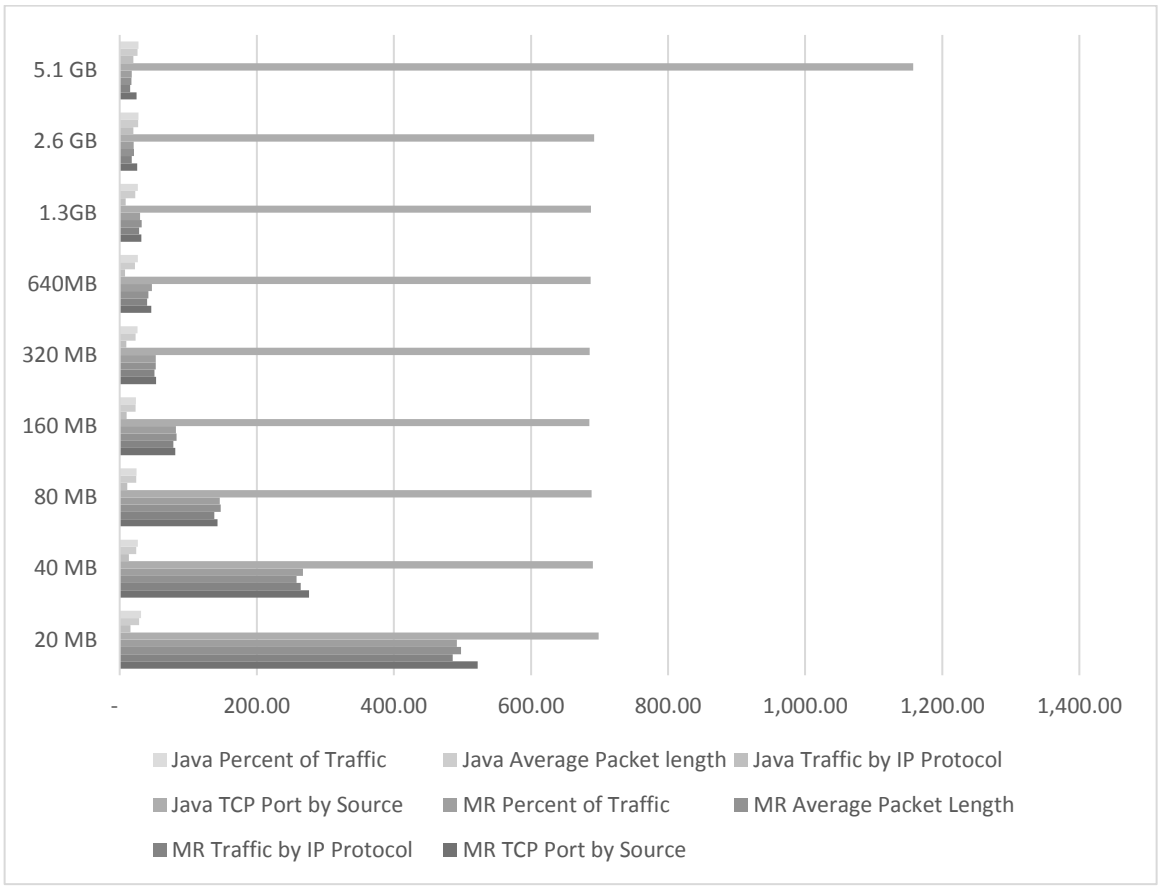


Figure 43. Time to process 50 TB in Days

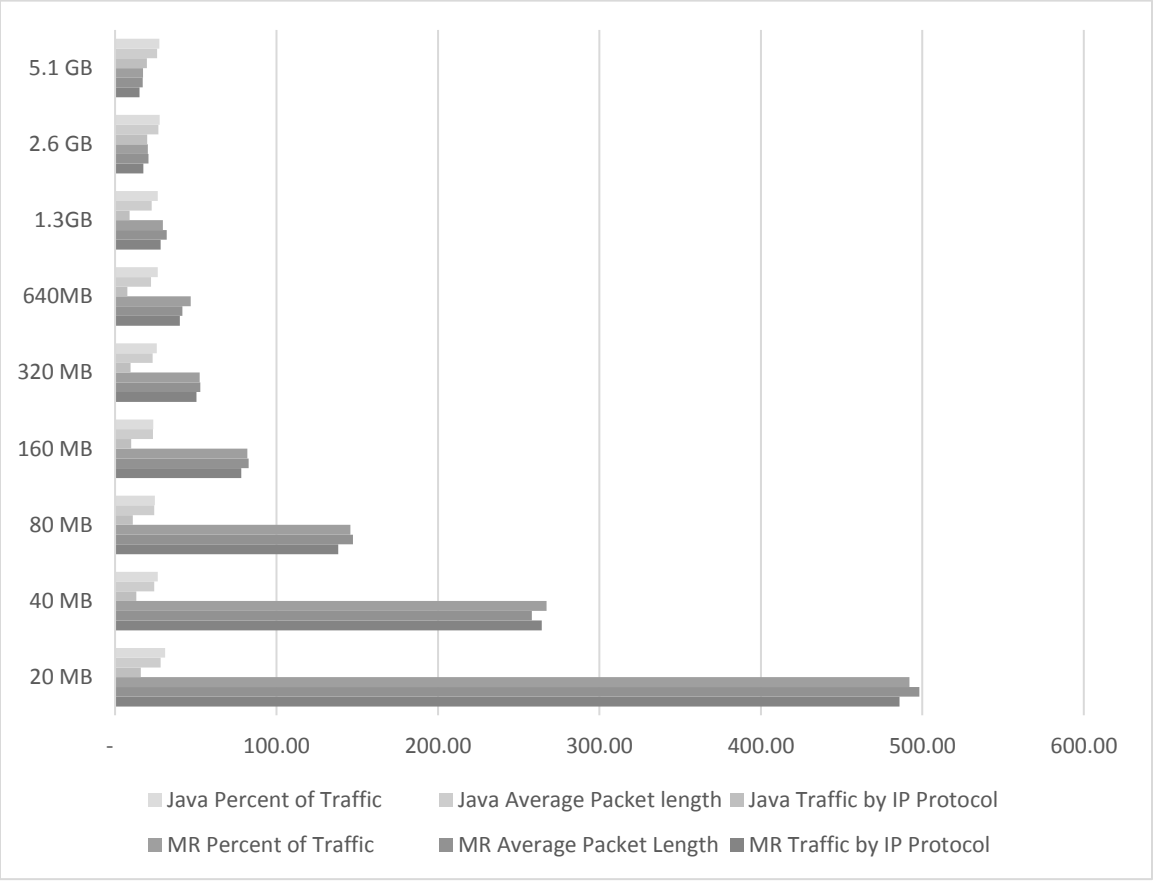


Figure 44. Time to process 50 TB in Days TCP Port by Source Removed

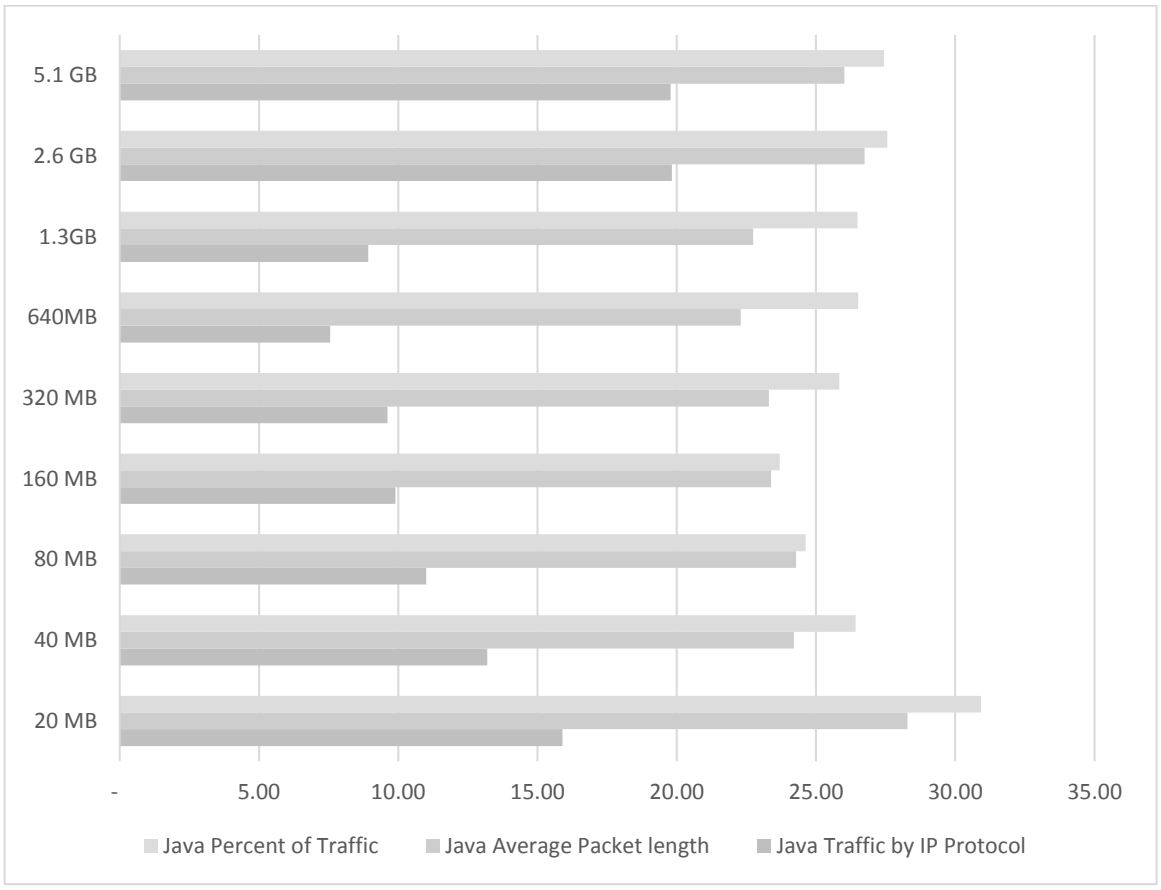


Figure 45. Time to process 50 TB in Days Java TCP Port by Source Removed

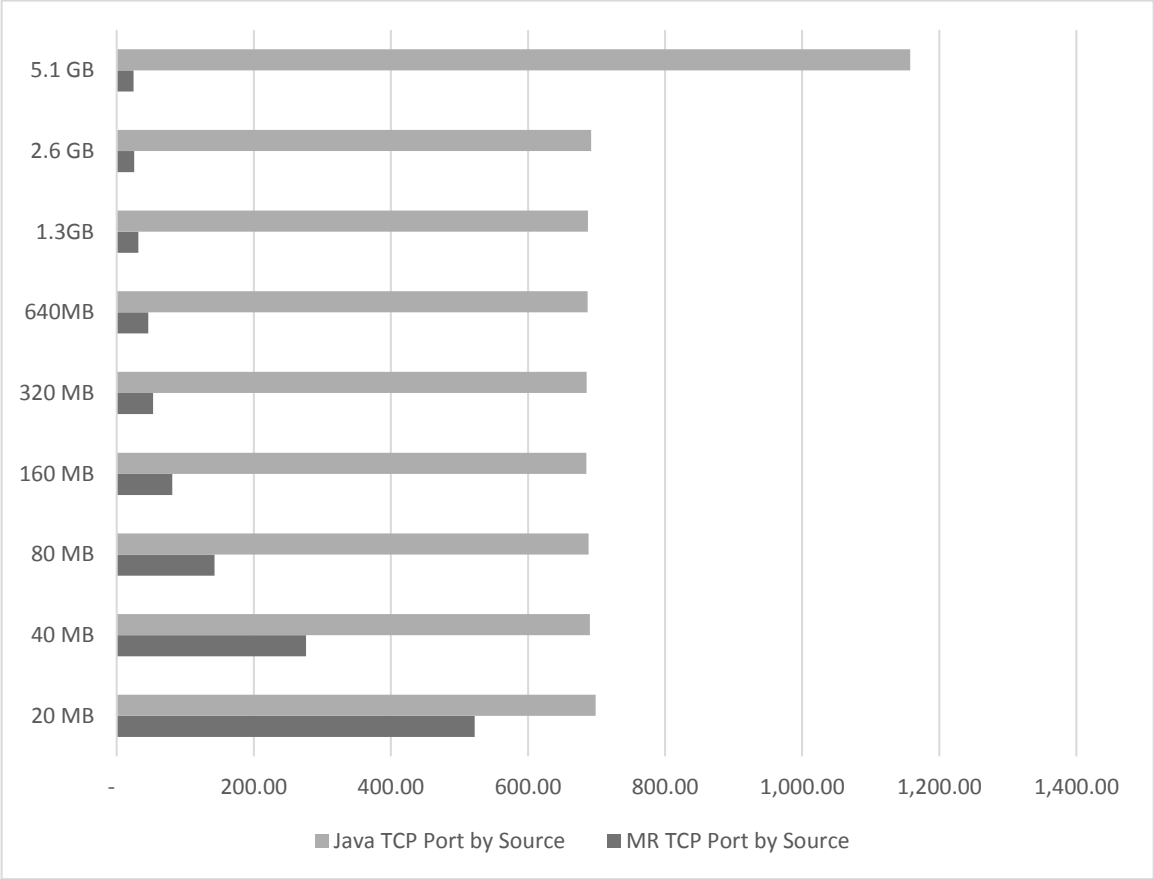


Figure 46. Time to process 50 TB in Days TCP Port by Source IP Data Excluded

Cost to Process (50 Terabytes)

The below values were calculated to determine the overall cost associated with each job using Amazon Web Services current pricing structure. The hourly CPU time and hourly storage cost were considered in the single node and clustered systems.

Table 13

AWS Cost Comparison

	CPU Hourly	CPU Monthly	Storage Hourly by GB Provisioned	Storage Monthly by GB Provisioned
Clustered	$0.48	$345.60	$0.000556	$0.001668
Single Node	$0.12	$86.40	$0.000139	$0.10

Source: AWS 2016

The below data represents the cost valuation for CPU and storage, in conjunction with the overall time it would take each program to process 50 terabytes of data.

Table 14

Total Cost to Process

	MR TCP Port by Source	MR Traffic by IP Protocol	MR Average Packet Length	MR Percent of Traffic	Java TCP Port by Source	Java Traffic by IP Protocol	Java Average Packet length	Java Percent of Traffic
20 MB	$90,730	$84,400	$86,510	$85,455	$39,773	$904	$1,610	$1,760
40 MB	$48,002	$45,892	$44,837	$46,420	$39,303	$751	$1,377	$1,504
80 MB	$24,792	$24,001	$25,583	$25,320	$39,183	$626	$1,382	$1,402
160 MB	$14,110	$13,583	$14,374	$14,242	$39,005	$563	$1,331	$1,349
320 MB	$9,231	$8,769	$9,165.	$9,099	$39,023	$547	$1,327	$1,471
640MB	$7,978	$6,956	$7,253.	$8,143	$39,109	$430	$1,269	$1,509
1.3GB	$5,489	$4,912	$5,538	$5,143	$39,153	$508	$1,295	$1,508
2.6 GB	$4,442	$3,041	$3,610	$3,535	$39,420	$1,128	$1,522	$1,568
5.1 GB	$4,248	$2,616	$2,971	$3,020	$65,910	$1,126	$1,481	$1,562

Calculation: Total Cost = (Hourly CPU cost*Hours to Complete)+(Hourly Storage Cost*To Complete * 544055 GBs)

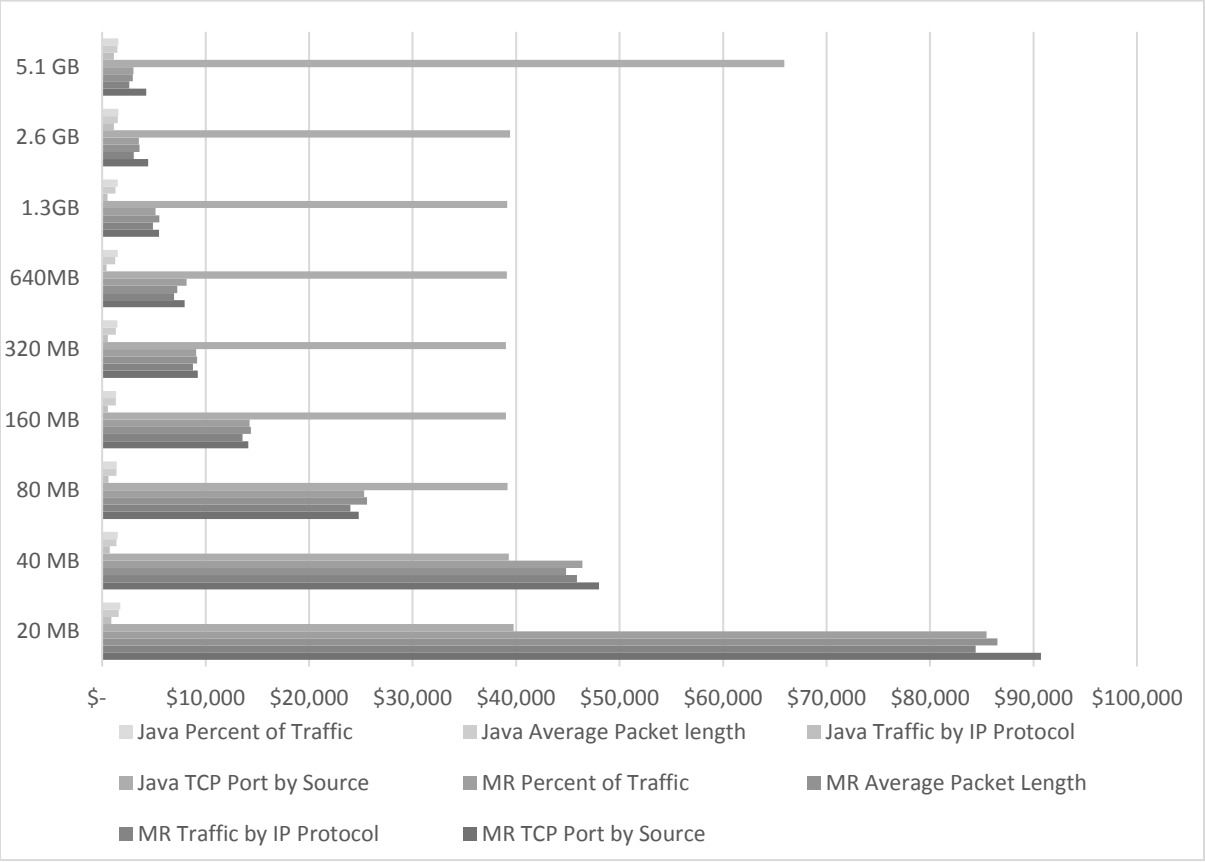
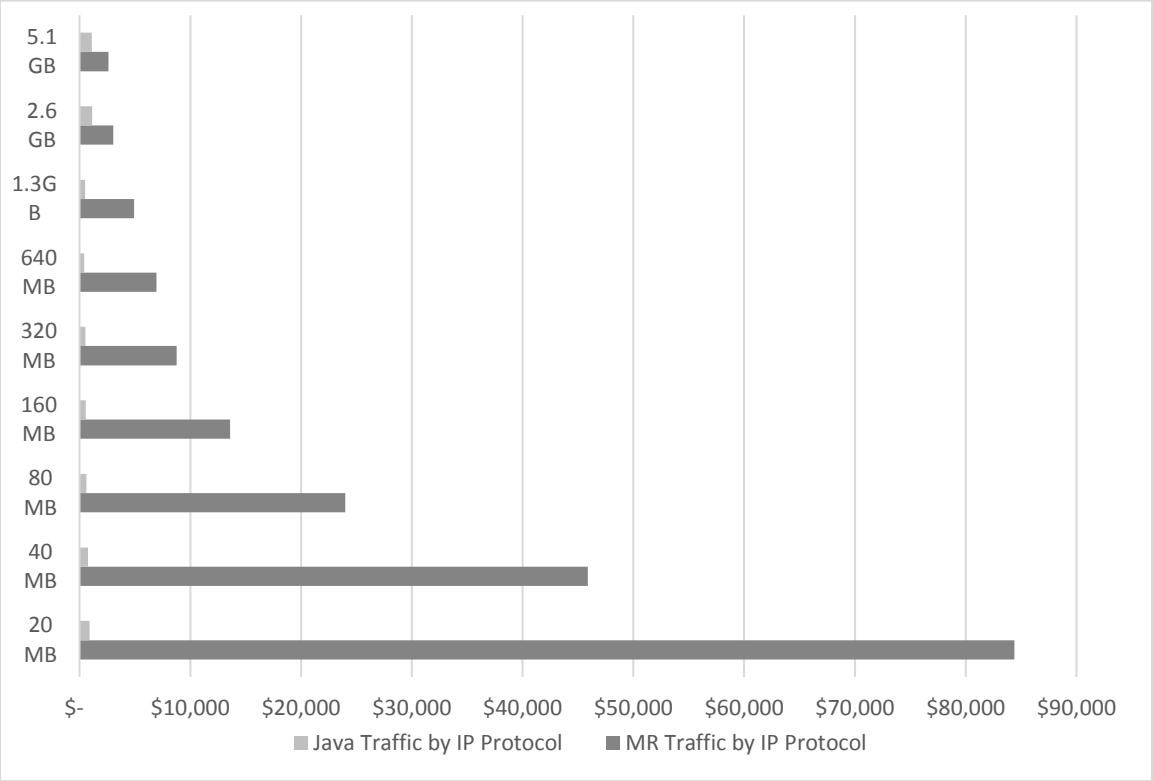


Figure 47. Cost to process 50 TB - All Programs



Figue 48. Cost to process 50 TB - Total Traffic by IP Protocol

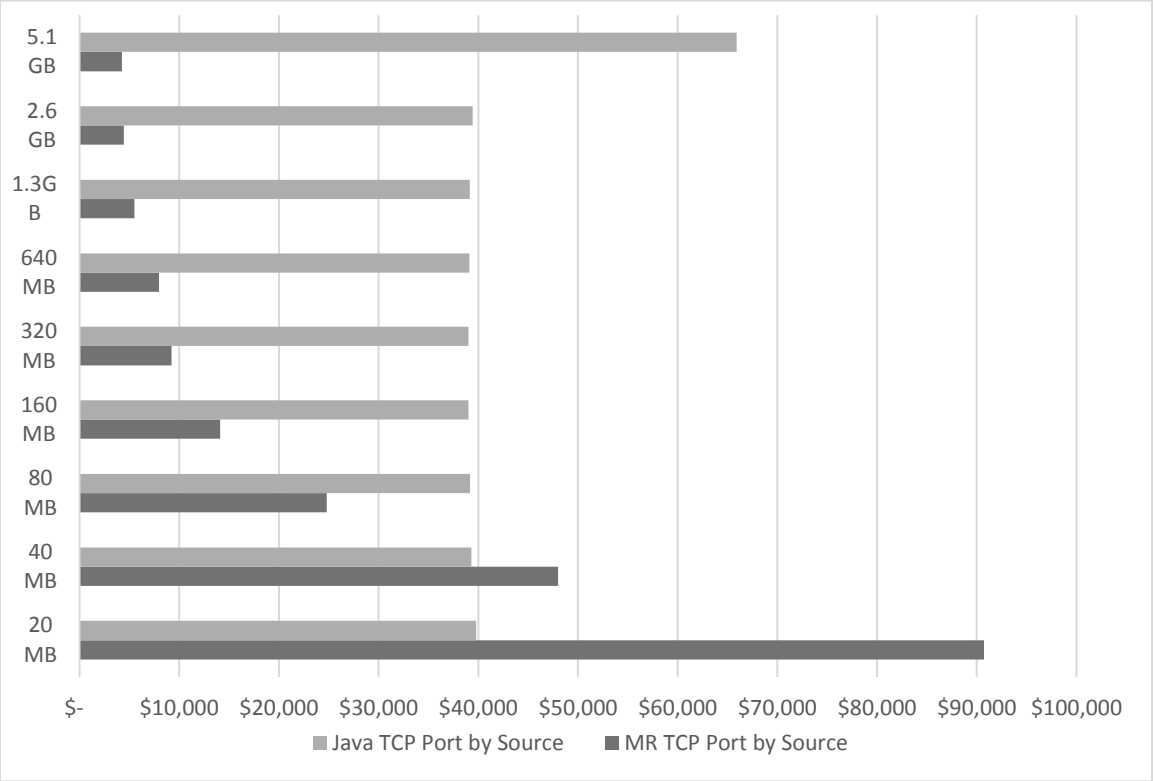


Figure 49. Cost to process 50 TB - TCP Port by Source IP

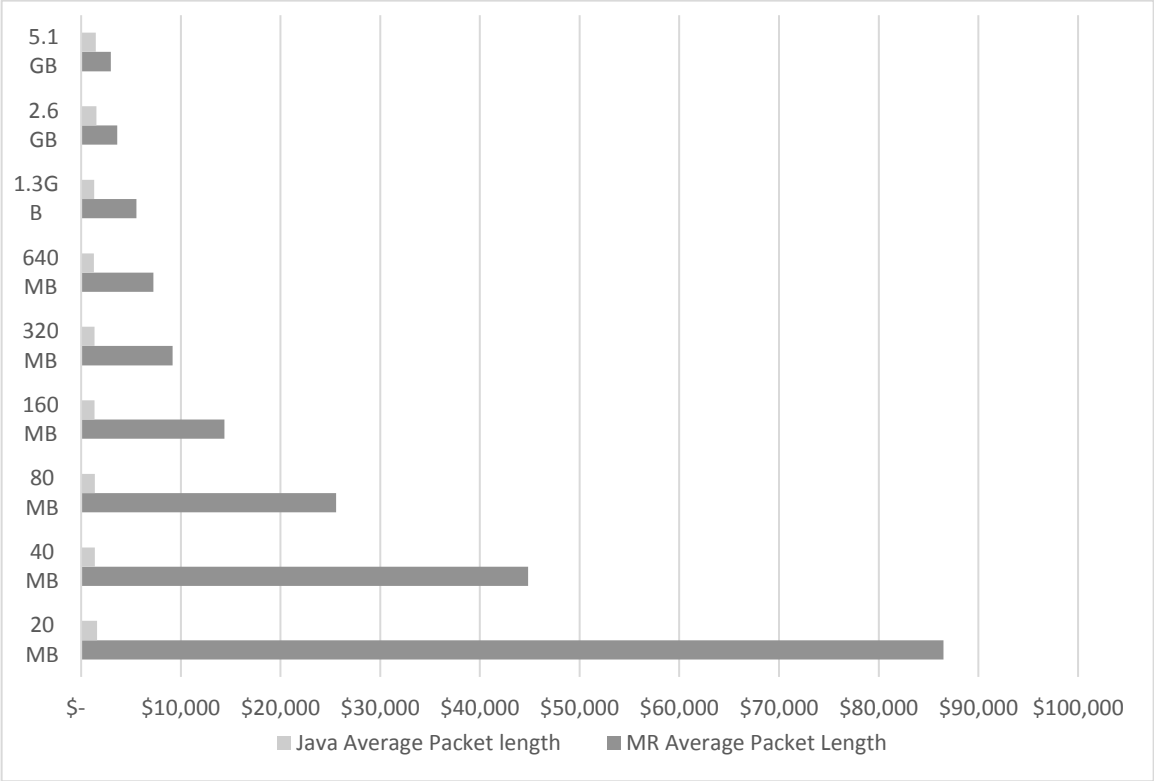


Figure 50. Cost to process 50 TB - Average Packet Length by IP

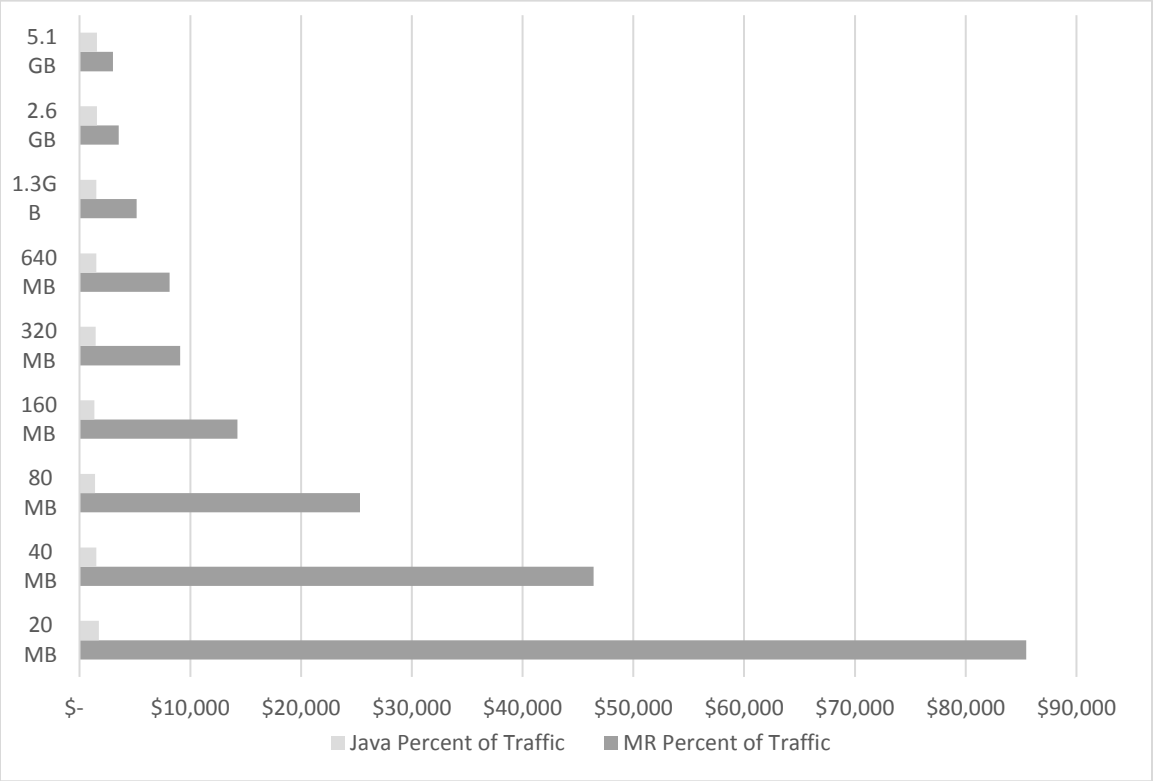


Figure 51. Cost to process 50 TB -Percentage of Traffic by IP

Data Analysis

An analysis of the data will be broken down into several sub-sections. The following sections will describe the specific behaviors and results of the Java and MapReduce programs. Prior to discussing the programs, several key components that lend to the integrity of the data acquired will be addressed.

AWS Resource Sharing

No identifiable days or times proved any consistent time differences throughout all 360 test runs. There were certain test runs that could be considered outliers due to a

difference of three or more seconds. These test did not correlate with any other program run. No distinct pattern could be determined to establish one specific day and time as negatively affecting the overall process. Any variations in time seem to fall on the overall memory availability of the processes in general, as there are no correlations to specific runs.

Hardware Limitations

Both the Java and MapReduce programs had abnormalities during initial testing. Java programs were set to collect packet length information in the form of bytes. This lead to memory issues when larger files were analyzed, as the Long and Double variables, reached their maximum values. This was corrected by mathematically converting the packet length variable to megabytes. Process time impact was negligible, and the Java programs ran with no further issues. MapReduce jobs proved problematic with memory in regards to the larger input files. On initial test runs of large data files, the overall process completed successfully, but specific map or reduce functions within each job failed due to Java heap space issues.

```
16/10/07 21:57:59 INFO mapreduce.Job: Task Id : attempt_1475870442079_0024_r_000000_0, Status : FAILED
Error: org.apache.hadoop.mapreduce.task.reduce.Shuffle$ShuffleError: error in shuffle in fetcher#5
        at org.apache.hadoop.mapreduce.task.reduce.Shuffle.run(Shuffle.java:134)
        at org.apache.hadoop.mapred.ReduceTask.run(ReduceTask.java:376)
        at org.apache.hadoop.mapred.YarnChild$2.run(YarnChild.java:164)
        at java.security.AccessController.doPrivileged(Native Method)
        at javax.security.auth.Subject.doAs(Subject.java:415)
        at org.apache.hadoop.security.UserGroupInformation.doAs(UserGroupInformation.java:1698)
        at org.apache.hadoop.mapred.YarnChild.main(YarnChild.java:158)
Caused by: java.lang.OutOfMemoryError: Java heap space
        at org.apache.hadoop.io.BoundedByteArrayOutputStream.<init>(BoundedByteArrayOutputStream.java:56)
        at org.apache.hadoop.io.BoundedByteArrayOutputStream.<init>(BoundedByteArrayOutputStream.java:46)
        at org.apache.hadoop.mapreduce.task.reduce.InMemoryMapOutput.<init>(InMemoryMapOutput.java:63)
        at org.apache.hadoop.mapreduce.task.reduce.MergeManagerImpl.unconditionalReserve(MergeManagerImpl.java:305)
        at org.apache.hadoop.mapreduce.task.reduce.MergeManagerImpl.reserve(MergeManagerImpl.java:295)
        at org.apache.hadoop.mapreduce.task.reduce.Fetcher.copyMapOutput(Fetcher.java:514)
        at org.apache.hadoop.mapreduce.task.reduce.Fetcher.copyFromHost(Fetcher.java:336)
        at org.apache.hadoop.mapreduce.task.reduce.Fetcher.run(Fetcher.java:193)
```

Figure 52. MapReduce Container Failure

Errors of this nature were corrected by modifying the maximum memory value asocated with containers within the hadoop configuration.

Job Process Output

All output from the Java and MapReduce jobs provided the same results. Due to the Java and MapReduce functions, output key values in different order, test runs for each output were imported to excel, sorted by the key value and then compared. A sample of the sorted data for the TCP Port to Source IP process is displayed below.

Table 15

Output Confirmation Example

MR Source IP	MR Port	MR Count	JavaSource IP	Java Port	Java Count	Match
10.101.21.172	55526	34624	10.101.21.172	55526	34624	YES
10.101.22.172	49452	1229152	10.101.22.172	49452	1229152	YES
104.107.22.19	80	2400	104.107.22.19	80	2400	YES
104.107.22.19	443	416	104.107.22.19	443	416	YES
104.107.38.239	443	640	104.107.38.239	443	640	YES
104.107.40.253	80	11488	104.107.40.253	80	11488	YES

Excel if statements were used to determine any inconsistencies between the two data sets.

Java/MapReduce Analysis

Analysis of test data results is broken down into two categories, number of unique keys and variable sizes within mathematical functions. These categories were the foundation when determining job output and must be revisited when performing data analysis.

Number of Unique Keys

The number of unique keys provided insight into how Java can be used to outperform MapReduce when searching for specific information. The Total Traffic by IP Protocol process had only two unique keys in the data that were used for testing. The Java Program only had to cycle through the input file a total of three times, one for initial input, and once for each unique key. The buffered reader input process outperformed the MapReduce function in every file size combination. The Java program would process the 50 terabytes of data faster with the 640MB files completing in 7.55 days. When calculating the estimated time with a larger file, the numbers increased dramatically, with the 5.1 GB file taking 19.72 days. The MapReduce function showed a steady decrease in processing time, with the 5.1 GB taking 15.07 days, two times slower than the Java Program.

In contrast to a low unique key value, the TCP Port by Source IP program must process through a high volume of unique keys because its need for an individual key for each IP address and Port combination. The input files used had a total of 1474 unique combinations. At no point could the Java process compete with the MapReduce. The Java code's best effort on 50 terabytes of data was an estimated 698 days, while the MapReduce process could complete this in 24.46. When the output values rely on a large quantity of unique keys, it is evident that MapReduce should be used.

Mathematical Functions

The second factor being tested, focused on mathematical functions containing different sized variables. In the Average Packet by Source IP and the Total Percentage of Traffic by Source IP programs, the unique key is kept the same, and relatively low at 375 IP addresses. In both programs performance increases as the file size of the input data grows larger. The change is notably larger when looking at the MapReduce code, with an estimated speed increase of 481 and 471 days for the average packet length and total percentage programs. The Java programs perform much better initially than the MapReduce with an estimated completion time of 28 and 30 days when using the file size at the 20 megabyte range for the average and percentage programs. The increase in efficiency does not match that of the MapReduce programs, with the 640MB file showing the best performance increase of only 6 and 4 days for the average and percentage programs.

Overall Cost Benefit Analysis

CPU and Storage cost were considered when determining which system would provide the most affordable option. The estimated costs of processing the 50 terabytes of information clearly shows the Java process as the superior method as long as the unique keys needed to perform the analysis remain low. In every instance of the Average Packet Length, Percentage of Traffic by IP, and Total Traffic by IP protocol programs, the Java code produced a significantly more cost effective process, cutting the cost by at least half compared to the same output obtained through MapReduce. The major cause of this resides in storage costs. As the amount of traffic analyzed increases, the MapReduce programs must replicate that data across at least three machines, possibly four depending upon cluster setup. Therefore the MapReduce process needs to makeup efficiency costs within the computational side in order to save overall costs by spending less on CPU cycles. When facing an analysis that requires a large number of unique keys, the MapReduce method offsets its data storage issue by being able to process the data at an exponentially higher rate. In the case of the TCP Port by Source IP program, we see the best run of the Java system costing $39005.19 while the MapReduce can provide the same output for $4248.85.

Summary

A clear understanding of the data collected by this research has been provided. A data set's overall size, the input file size, and the number of unique values needed for analysis impact the performance of both Java and MapReduce-based systems. In the

next chapter, the impact of these distinct variables and how they should be questioned and analyzed will be reviewed. The original hypothese from the beginning of this paper will also be revisited.

Chapter V

RESULTS, CONCLUSION, AND RECOMMENDATIONS

Introduction

The final section of this document will provide a summation of the work in its entirety. A brief overview of the methodology and results will be provided. This information will be used to revisit the initial hypotheses to explore if the initial questions have been answered. Lastly, possible future work will be discussed as it relates to opportunities created by the research discussed within this paper.

Results

In an effort to determine the most effective and cost efficient option for network log analysis, a comparison between MapReduce jobs running on a four cluster HDFS AWS system and custom Java based jobs running on a single AWS server was performed. Four separate outputs were intended to provide the proper test cases, with MapReduce and Java jobs used to create identical output. Two possible components were considered when creating the code, the number of unique keys and the use of large values within mathematical functions. The unique key value deemed to be the most important. Analysis that required a low volume of unique values provided a large efficiency gain for the Java based code. A cost benefit analysis was done that determined the Java code was more cost effective in all tests except those with a high

number of unique keys. In these tests, the MapReduce job performed the same task twenty-nine times faster and with a cost $34,756 lower than the most efficient Java version. With the information gathered from the above research, the original hypotheses questions can be revisited.

1) What is the performance increase in utilizing native Java versus clustered MapReduce on varying sized data sets?

   The data provided by this research shows that Java-based programs outperform the MapReduce processes when a small amount of data is being analyzed. This conclusion is supported by the limited ability MapReduce has when handling small files. In this instance MapReduce is more efficient with a small number of large files. The overhead of dividing the job workload to several systems outperformed the Java at the 2.6GB file mark.

2) What is the specific data set size MapReduce provides enough benefit to warrant the extra cost of the hardware?

   No specific data set size was found to provide a dramatic performance increase within the MapReduce programs. Instead, each larger dataset saw improved performance gains. This is in contrast to the Java-based code that could be seen reaching its optimal processing times at the 640MB dataset. It should also be noted that the Java-based programs outperformed the MapReduce on all datasets when a small number of unique keys was required.

3) Can cloud computing CPU clusters be utilized to offset the higher cost of hardware-based clustered services?

No definitive answer to this question was discovered within the research, but the ability to customize a system to run based upon the number of unique keys could provide a large amount cost savings. The research proves that a java program intended to search for a small number of unique keys can greatly outperform a MapReduce function of the same design. This would prove beneficial when searching for a specific root cause of attacks when the source vector is known, and a large volume of data is under examination.

Conclusion

In conclusion, this report provides proof that custom Java code is more cost and time efficient than MapReduce code, when performing with a volume of unique keys below 400. MapReduce technology can greatly improve the efficiency of log analysis but it is not the tool to be used in every scenario. As network traffic continues to increase, the most efficient means of obtaining the necessary information within this report proves it is not the newest technology.

Future work

The research could be further expanded by testing larger datasets and utilizing more powerful AWS cloud servers to establish a wider range of data sets. The number of unique keys within the data could also be explored to establish efficiency of Java and

MapReduce over a more granular unique key metric. Lastly, as the Aparapi API continues to evolve, MapReduce and Java implementation in this space should be revisited.

REFERENCES

Ahmed, M., Mahmood, A., Hu, J. (2013). Outlier Detection. The State of the Art in Intrusion Prevention and Detection, 3-22.

Albin, E., & Rowe, N. C. (2011). A Realistic Experimental Comparison of the Suricata and Snort Intrusion-Detection Systems. 2012 26th International Conference on Advanced Information Networking and Applications Workshops. Retrieved March 22, 2016, from http://www.dtic.mil/dtic/tr/fulltext/u2/a552115.pdf

Apache Spark - Lightning-Fast Cluster Computing. (2016). Retrieved March 24, 2016, from http://spark.apache.org/

An introduction to Apache Hadoop for big data. (2014). Retrieved March 25, 2016, from https://opensource.com/life/14/8/intro-apache-hadoop-big-data

Cheon, J., & Choe, T. (2013). Distributed Processing of Snort Alert Log using Hadoop. International Journal of Engineering & Technology, 3(3), 2685. Retrieved March 23, 2016, from http://www.enggjournals.com/ijet/docs/IJET13-05-03-178.pdf

Dean, J., & Ghemawat, S. (2004). MapReduce. Communications of the ACM Commun. ACM, 51(1), 107. Retrieved March 23, 2016, from http://research.google.com/archive/mapreduce.html

EC2 Instance Types – Amazon Web Services (AWS). (2016). Retrieved March 24, 2016, from https://aws.amazon.com/ec2/instance-types/

El Jamiy, F., Daif, A., Azouazi, M., & Marzak, A. (2014). The potential and challenges of Big data - Recommendation systems next level application. International Journal

of Computer Science Issues, 11(5), 21. Retrieved March 23, 2016, from http://arxiv.org/ftp/arxiv/papers/1501/1501.03424.pdf

Global Cloud Index (GCI). (2015, October). Retrieved March 18, 2016, from http://www.cisco.com/c/en/us/solutions/service-provider/global-cloud-index-gci/index.html

Gartner. (2015, November). Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015 [Press release]. Retrieved March 23, 2016, from http://www.gartner.com/newsroom/id/3165317

Jiang, W., & Prasanna, V. (2013). Hardware Techniques for High-Performance Network Intrusion Detection. The State of the Art in Intrusion Prevention and Detection, 233-256.

Joshi, S. (2012). Leveraging Aparapi to Help Improve Java Financial ... Retrieved March 22, 2016, from http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/Leveraging_Aparapi_to_Improve_Java_Financial_Application_Performance_after_legal_review.pdf

Mell, P. M., & Grance, T. (2011). The NIST definition of cloud computing. Retrieved March 22, 2016, from http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

Olavsrud, T. (2012, July 31). Can Big Data Help Universities Tackle Security, BYOD? Retrieved March 23, 2016, from http://www.cio.com/article/2393636/education/can-big-data-help-universities-tackle-security--byod-.html

Pranggono, B., Mclaughlin, K., Yang, Y., & Sezer, S. (2013). Intrusion Detection Systems for Critical Infrastructure. The State of the Art in Intrusion Prevention and Detection, 115-138.

Scarfone, K. A., & Mell, P. M. (2007). Guide to Intrusion Detection and Prevention Systems (IDPS). Retrieved March 22, 2016, from http://csrc.nist.gov/publications/nistpubs/800-94/SP800-94.pdf

Thosar, S., Mane, A., Raykar, S., Jain, R., Khude, P., & Guru, S. (2015). Survey on Log Analysis and Management. International Journal of Computer Science Trends and Technology (IJCST), 3(6), 9-14. Retrieved March 22, 2016, from http://www.ijcstjournal.org/volume-3/issue-6/IJCST-V3I6P2.pdf

Vasiliadis, G., Antonatos, S., Polychronakis, M., Markatos, E. P., & Ioannidis, S. (2008). Gnort: High Performance Network Intrusion Detection Using Graphics Processors. Lecture Notes in Computer Science Recent Advances in Intrusion Detection, 116-134. Retrieved March 22, 2016, from http://users.ics.forth.gr/~sotiris/publications/conference/32raid2008.pdf

Appendix A: Java TCP Port by Source IP

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.text.DecimalFormat;
import java.util.HashSet;



public class JavaSourcePortMap{

	public static  void main (String[] args) throws Throwable
	{
		long StartTime = System.currentTimeMillis();
		long StopTime;
		long elapsedTime;
		String line;

		DecimalFormat df = new DecimalFormat();
		df.setMaximumFractionDigits(4);

		File file=new File(args[1]);

		HashSet<String> uniqueLine = new HashSet<String>(); //Unique has to
obtain unique  values

		 //Reads each line from file into an array list
		BufferedReader reader = new BufferedReader(new FileReader(args[0]));

		while ((line = reader.readLine()) !=null)
		{
		   String temp[] = line.toString().split("\t"); //Reads each line of
input file and splits it by tab into a temp array
		    if (temp[1].contains("6"))
		       uniqueLine.add(temp[2]+ " " + temp[3]); //adds IP source and Port
as a unique value to hash set
		}
		   reader.close();

		   String[] uniqueArray = uniqueLine.toArray(new
String[uniqueLine.size()]);
		   int[] countArray = new int[uniqueLine.size()];

		//Collection function that acts as reducer phase for array list

		 if(!file.exists())
			file.createNewFile();

		FileWriter fw = new FileWriter(file.getAbsoluteFile());
```

```
            BufferedWriter bw = new BufferedWriter(fw);

            BufferedReader input_reader = new BufferedReader(new
FileReader(args[0]));
              while ((line = input_reader.readLine()) !=null){
                    String temp[] = line.toString().split("\t"); //Reads each line
of input file and splits it by tab into a temp array

                    //For loop counts the number of instances of each unique value
and increments corresponding array location
                    for(int j=0;j<uniqueArray.length;j++){
                          if((temp[2]+ " " + temp[3]).equals(uniqueArray[j])){
                                countArray[j]++;
                               }
                          }
              }
              input_reader.close();

             for(int i=0;i<uniqueArray.length;i++)
               bw.write(uniqueArray[i] + " " + countArray[i]  + "\n");

             StopTime = System.currentTimeMillis();
             elapsedTime = StopTime - StartTime; //calculates program run time
             bw.write("Total elapsed time in milliseconds: " + elapsedTime + "\n");
             bw.close();
       }
                                        }
```

Appendix B: Java Protocol Traffic Source Code

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.text.DecimalFormat;
import java.util.HashSet;


public class JavaProtocolThroughput{

	public static void main (String[] args) throws Throwable
	{
		long StartTime = System.currentTimeMillis();
		long StopTime;
		long elapsedTime;
		String line;

		DecimalFormat df = new DecimalFormat();
		df.setMaximumFractionDigits(4);

		File file=new File(args[1]);

		HashSet<String> uniqueProtocol = new HashSet<String>(); //Unique has to
obtain unique protocol values

		//Reads each line from file into an array list
	    BufferedReader reader = new BufferedReader(new FileReader(args[0]));

	    while ((line = reader.readLine()) !=null)
	    {
	       String temp[] = line.toString().split("\t"); //Reads each line of input
file and splits it by tab into atemp array
		uniqueProtocol.add(temp[1]); //Adds only unique protocol values into
the Unique HashSet uniqueIPs
		}
		   reader.close();

		   String[] ProtocolArray = uniqueProtocol.toArray(new
String[uniqueProtocol.size()]);
		   float[] sumArray = new float[uniqueProtocol.size()];


		//Collection function that acts as reducer phase for array list

		 if(!file.exists())
			file.createNewFile();

		FileWriter fw = new FileWriter(file.getAbsoluteFile());
		BufferedWriter bw = new BufferedWriter(fw);
```

```
              BufferedReader input_reader = new BufferedReader(new
FileReader(args[0]));
                while ((line = input_reader.readLine()) !=null){
                       float k;
                       String temp[] = line.toString().split("\t");

                       //for loop searches for each unique protocol number and
increments traffic for the corresponding array
                       for(int j=0;j<ProtocolArray.length;j++){
                              if(temp[1].equals(ProtocolArray[j])){
                                     k= Float.parseFloat(temp[7]);
                                     sumArray[j]+=(k/1024)/1024;
                                     }
                              }
                }
                input_reader.close();

                for(int i=0;i<ProtocolArray.length;i++)
                {
                 bw.write("Protocol = " + ProtocolArray[i] + " Total = " +
sumArray[i] + " megabytes\n");
                }

               StopTime = System.currentTimeMillis();
               elapsedTime = StopTime - StartTime;
               bw.write("Total elapsed time in milliseconds: " + elapsedTime + "\n");
               bw.close();

}
}
```

Appendix C: Java Average Packet Length by Source IP Source Code

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.text.DecimalFormat;
import java.util.HashSet;

public class JavaAveragePacketLen{
       public static void main (String[] args) throws Throwable
       {

          long StartTime = System.currentTimeMillis();
          long StopTime;
          long elapsedTime;

          String line;

          DecimalFormat df = new DecimalFormat();
          df.setMaximumFractionDigits(4);

          File file=new File(args[1]);

          HashSet<String> uniqueIPs = new HashSet<String>();

          //Reads each line from file into an array list
        BufferedReader reader = new BufferedReader(new FileReader(args[0]));

        while ((line = reader.readLine()) !=null)
        {
          String temp[] = line.toString().split("\t");  //Reads each line of input
file and splits it by tab into a temp array
            uniqueIPs.add(temp[2]);   //Adds only unique IP addresses into the Unique
HashSet uniqueIPs
        }
          reader.close();

          String[] IPArray = uniqueIPs.toArray(new String[uniqueIPs.size()]);
//Converts IPs to array of strings
          float[] sumArray = new float[uniqueIPs.size()]; //Array that will
correspond with IPArray for store total traffic per IP
          float[] countArray = new float[uniqueIPs.size()]; //Array that will
correspond with IPArray to store total number of packets per IP

          //Collection function that acts as reducer phase for array list

          if(!file.exists())
                file.createNewFile();
```

```
          FileWriter fw = new FileWriter(file.getAbsoluteFile());
          BufferedWriter bw = new BufferedWriter(fw);

          BufferedReader input_reader = new BufferedReader(new FileReader(args[0]));
            while ((line = input_reader.readLine()) !=null){
                  float k=0;
                  String temp[] = line.toString().split("\t");//Reads each line of
input file and splits it by tab into a temp array

                  //For Loop below checks each input line for the matching IP from
the uniqueIP array and increments the corresponding sum and count arrays
                  for(int j=0;j<uniqueIPs.size();j++){
                       if(temp[2].equals(IPArray[j])){
                             k= Float.parseFloat(temp[7]);
                             sumArray[j]+=k;
                             countArray[j]++;
                             }
                       }
          }
          input_reader.close();

          //For loop cycles through each array,and prints the total traffic from sum,
and calculates the average for each unique IP Addres
          for(int i=0;i<uniqueIPs.size();i++)
          {
           bw.write(IPArray[i] + "\tTotal =\t" + sumArray[i] + "\tAvg =\t" +
(sumArray[i]/countArray[i]) + "\n");
          }

           StopTime = System.currentTimeMillis();
           elapsedTime = StopTime - StartTime;   //calculates runtime of program
           bw.write("Total elapsed time in milliseconds: " + elapsedTime + "\n");
//Prints output to file
           bw.close();
      }
}
```

Appendix D: Java Total Percentage of Traffic by IP Source Code

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.text.DecimalFormat;
import java.util.HashSet;


public class JavaPercentageOfTraffic{
	static float Total_traffic = 0;  //Static value holds total traffic to be used
in percentage calculation
	public static void main (String[] args) throws Throwable
	{
		long StartTime = System.currentTimeMillis();
		long StopTime;
		long elapsedTime;
		String line;

		DecimalFormat df = new DecimalFormat();
		df.setMaximumFractionDigits(6);

		File file=new File(args[1]);

		HashSet<String> uniqueIPs = new HashSet<String>();  //Unique has to
obtain unique IP values

		//Reads each line from file into an array list
	    BufferedReader reader = new BufferedReader(new FileReader(args[0]));

	    while ((line = reader.readLine()) !=null)
	    {
		    String temp[] = line.toString().split("\t"); //Reads each line of
input file and splits it by tab into a temp array
		    uniqueIPs.add(temp[2]);  //Adds only unique IP addresses into the
Unique HashSet uniqueIPs
	    }
		reader.close();

		String[] IPArray = uniqueIPs.toArray(new String[uniqueIPs.size()]);
//Converts IPs to array of strings
		float[] sumArray = new float[uniqueIPs.size()]; //Array that will
correspond with IPArray for store total traffic per IP

		if(!file.exists())
		  file.createNewFile();

		FileWriter fw = new FileWriter(file.getAbsoluteFile());
		BufferedWriter bw = new BufferedWriter(fw);
```

```
            BufferedReader input_reader = new BufferedReader(new
FileReader(args[0]));
            while ((line = input_reader.readLine()) !=null)
            {
               String temp[] = line.toString().split("\t"); //Reads each line of
input file and splits it by tab into a temp array

               //For loop compares each input line to the unique IP and incrememnts
sum and total when applicatble
               for(int j=0;j<IPArray.length;j++)
               {
               if(temp[2].equals(IPArray[j]))
               {
                     sumArray[j]+=(Float.parseFloat(temp[7])/1024)/1024;  //adds to
sum of unique IPs traffic
                     Total_traffic+=(Float.parseFloat(temp[7])/1024)/1024;   //Adds
to total traffic
               }
               }
            }
            input_reader.close();


            for(int i=0;i<IPArray.length;i++)
            {
               float percent;
               percent = (((sumArray[i]/Total_traffic)*100));
               if(percent <.001)
               bw.write(IPArray[i] +"\t " +  df.format(percent) + "\n"); //generates
output
            }

            StopTime = System.currentTimeMillis();
            elapsedTime = StopTime - StartTime;  //calculates program run time
            bw.write("Total elapsed time in milliseconds: " + elapsedTime + "\n");
            bw.close();
         }
}
```

Appendix E: MapReduce TCP Port by Source IP Source Code

```
import java.io.IOException;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class SourcePortMap {

 public static class Map extends Mapper<LongWritable, Text, Text, IntWritable>
 {

    String out_map; //String to store unique key value for mapping

    public void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException
    {
        String temp[] = value.toString().split("\t"); //Reads each line of input file
into an array split by tabs
        if (temp[1].contains("6"))
             out_map = temp[2] + " " + temp[3];  //Looks for only protocol 6 data and
sets each mapped value to SourceIP Port
        context.write(new Text(out_map),new IntWritable(1)); //Sets integer at 1
value for each output of the Source Port combination
     }
 }

 public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable>
{

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values)
        {
            sum += val.get(); //stores total number of each individual key
incrementing off of the mapped integer of 1
        }
        context.write(key, new IntWritable(sum));
    }
 }

 public static void main(String[] args) throws Exception
 {
    Configuration conf = new Configuration();
```

```
        @SuppressWarnings("deprecation")
             Job job = new Job(conf, "SourcePortMap");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    job.setJarByClass(SourcePortMap.class);


    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
 }

}
```

Appendix F: MapReduce Protocol Traffic Source Code

```
import java.io.IOException;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class ProtocolThroughput {

 public static class Map extends Mapper<LongWritable, Text, Text, FloatWritable>
 {
      private FloatWritable output = new FloatWritable();

    public void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException
    {
        String temp[] = value.toString().split("\t"); //Reads each line of input file
into an array split by tabs
        output.set(Float.parseFloat(temp[7])); //Sets the output value to the packet
length of the input line as an integer
        context.write(new Text(temp[1]),output);      //Sets a mapping of the
protocol number to the the integer value of the packet length
    }
 }
 public static class Reduce extends Reducer<Text, FloatWritable, Text, FloatWritable>
{

    public void reduce(Text key, Iterable<FloatWritable> values, Context context)
      throws IOException, InterruptedException
    {
        float sum = 0;
        for (FloatWritable val : values)
        {
            sum += (val.get()/1024)/1024;  //sums the total traffic in bytes for each
unique protocol number
        }
        context.write(key, new FloatWritable(sum));  //Writes output of protocol
number and total traffic
    }
 }

 public static void main(String[] args) throws Exception
 {
    Configuration conf = new Configuration();

    @SuppressWarnings("deprecation")
```

```
        Job job = new Job(conf, "ProtocolThroughput");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(FloatWritable.class);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    job.setJarByClass(ProtocolThroughput.class);


    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
 }

}
```

Appendix G: MapReduce Average Packet Length by Source IP Source Code

```
import java.io.IOException;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class AveragePacketLen {

 public static class Map extends Mapper<LongWritable, Text, Text, FloatWritable>
 {
    private FloatWritable output = new FloatWritable();

    public void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException
    {
        String temp[] = value.toString().split("\t"); //Reads each line of input file
into an array split by tabs
        output.set(Float.parseFloat(temp[7]));        //Sets the output value of the
map to the packet length
        context.write(new Text(temp[2]),output);      //Outputs a mapping of the
source IP and the packet length
     }
 }

 public static class Reduce extends Reducer<Text, FloatWritable, Text, FloatWritable>
{
    public void reduce(Text key, Iterable<FloatWritable> values, Context context)
      throws IOException, InterruptedException
      {
        float sum = 0;   // value to hold total traffic for each unique IP
        float avg = 0;   // variable to hold average value
        float count = 0; // count variable to determine how many packets for each
unique IP
        for (FloatWritable val : values)
        {
            sum += val.get();      //sums the value of each unique IP
                   count++;;      //increments counter to be used in average
calculation
        }
        avg = sum/count;           //calculates average
        context.write(key, new FloatWritable(avg)); //provides output mapping
     }
 }

 public static void main(String[] args) throws Exception
 {
```

```
    Configuration conf = new Configuration();

    @SuppressWarnings("deprecation")
    Job job = new Job(conf, "AveragePacketLen");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(FloatWritable.class);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    job.setJarByClass(AveragePacketLen.class);


    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
 }
}
```

Appendix H: MapReduce Total Percentage of Traffic by IP Source Code

```
import java.io.IOException;
import java.text.DecimalFormat;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class PercentageOfTraffic {

static float Total_traffic = 0; //Stores total traffic for packets in input file

 public static class Map extends Mapper<LongWritable, Text, Text, FloatWritable>
 {
    //private FloatWritable output = new FloatWritable();

    public void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException
    {

        String temp[] = value.toString().split("\t"); //Reads each line of input file
into an array split by tabs
        float val_temp = Float.parseFloat(temp[7]);
        val_temp = (val_temp/1024)/1024;

        Total_traffic+=val_temp;     //increments the total traffic value
        context.write(new Text(temp[2]),new FloatWritable(val_temp));      //maps
each source iP to a traffic value
     }
 }
 public static class Reduce extends Reducer<Text, FloatWritable, Text, FloatWritable>
{

    public void reduce(Text key, Iterable<FloatWritable> values, Context context)
      throws IOException, InterruptedException
    {
          DecimalFormat df = new DecimalFormat();
          df.setMaximumFractionDigits(6);
        float sum = 0;


        for (FloatWritable val : values)
        {
            sum += val.get();   //sums total traffic for each unique IP
        }
```

```
        context.write(key, new FloatWritable(((sum/Total_traffic)*100))); //returns
string format in order to limit number of decimals
    }
 }

 public static void main(String[] args) throws Exception
 {

    Configuration conf = new Configuration();

    @SuppressWarnings("deprecation")
    Job job = new Job(conf, "PercentageOfTraffic");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(FloatWritable.class);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    job.setJarByClass(PercentageOfTraffic.class);


    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
 }

}
```

Appendix I – MapReduce TCP Port by Source IP Test Data

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	1:45:00 AM	17	19.98
Test 2	5-Oct	3:24:00 PM	17	19.98
Test 3	6-Oct	10:16:00 AM	17	19.98
Test 4	9-Oct	1:44:00 PM	17	19.98
Test 5	15-Oct	10:53:00 PM	18	19.98
Avg			17.2	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	1:45:00 AM	19	39.96
Test 2	5-Oct	3:24:00 PM	18	39.96
Test 3	6-Oct	10:15:00 AM	18	39.96
Test 4	9-Oct	1:43:00 PM	18	39.96
Test 5	15-Oct	10:53:00 PM	18	39.96
Avg			18.2	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	1:45:00 AM	19	79.92
Test 2	5-Oct	3:24:00 PM	19	79.92
Test 3	6-Oct	10:15:00 AM	19	79.92
Test 4	9-Oct	1:43:00 PM	19	79.92
Test 5	15-Oct	10:54:00 AM	18	79.92
Avg			18.8	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	1:44:00 AM	21	159.84
Test 2	5-Oct	3:23:00 PM	22	159.84
Test 3	6-Oct	10:14:00 AM	22	159.84
Test 4	9-Oct	1:42:00 PM	22	159.84
Test 5	15-Oct	10:54:00 PM	20	159.84
Avg			21.4	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	1:44:00 AM	29	319.68
Test 2	5-Oct	3:23:00 PM	28	319.68
Test 3	6-Oct	10:14:00 AM	28	319.68
Test 4	9-Oct	1:42:00 PM	28	319.68
Test 5	15-Oct	10:55:00 PM	27	319.68
Avg			28	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	1:43:00 AM	46	639.36
Test 2	5-Oct	3:22:00 PM	50	639.36
Test 3	6-Oct	10:13:00 AM	48	639.36
Test 4	9-Oct	1:41:00 PM	47	639.36
Test 5	15-Oct	10:56:00 AM	51	639.36
Avg			48.4	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	1:42:00 AM	70	1278.72
Test 2	5-Oct	3:21:00 PM	58	1278.72
Test 3	6-Oct	10:12:00 AM	69	1278.72
Test 4	9-Oct	1:40:00 PM	69	1278.72
Test 5	15-Oct	10:57:00 PM	67	1278.72
Avg			66.6	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	1:39:00 AM	108	2557.44
Test 2	5-Oct	3:19:00 PM	108	2557.44
Test 3	6-Oct	10:09:00 AM	107	2557.44
Test 4	9-Oct	1:38:00 PM	109	2557.44
Test 5	15-Oct	10:59:00 AM	107	2557.44
Avg			107.8	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	1:35:00 AM	206	5115
Test 2	5-Oct	3:15:00 PM	208	5115
Test 3	6-Oct	10:05:00 AM	205	5115
Test 4	9-Oct	1:34:00 PM	206	5115
Test 5	15-Oct	11:00:00 PM	206	5115
Avg			206.2	

Appendix J – MapReduce Total Traffic by IP Protocol Tests

	Date	Start Time	CPU Time	File Size
Test 1	30-Sep	1:52 AM	16	19.98
Test 2	1-Oct	3:31 PM	15	19.98
Test 3	2-Oct	10:24 AM	16	19.98
Test 4	3-Oct	1:51 PM	17	19.98
Test 5	15-Oct	11:32 PM	16	19.98
Avg			16	

	Date	Start Time	CPU Time	File Size
Test 1	30-Sep	1:52 AM	17	39.96
Test 2	1-Oct	3:30 PM	18	39.96
Test 3	2-Oct	10:24 AM	18	39.96
Test 4	3-Oct	1:51 PM	18	39.96
Test 5	15-Oct	11:32 PM	16	39.96
Avg			17.4	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	1:52 AM	20	79.92
Test 2	5-Oct	3:30 PM	19	79.92
Test 3	6-Oct	10:24 AM	17	79.92
Test 4	9-Oct	1:51 AM	18	79.92
Test 5	15-Oct	11:33 PM	17	79.92
Avg			18.2	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	1:51 AM	22	159.84
Test 2	5-Oct	3:30 PM	21	159.84
Test 3	6-Oct	10:23 AM	21	159.84
Test 4	9-Oct	1:50 PM	20	159.84
Test 5	15-Oct	11:34 AM	19	159.84
Avg			20.6	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	1:51 AM	27	319.68
Test 2	5-Oct	3:29 PM	26	319.68
Test 3	6-Oct	10:23 AM	28	319.68
Test 4	9-Oct	1:50 PM	28	319.68
Test 5	15-Oct	11:34 AM	24	319.68
Avg			26.6	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	1:50 AM	42	639.36
Test 2	5-Oct	3:29 PM	43	639.36
Test 3	6-Oct	10:22 AM	42	639.36
Test 4	9-Oct	1:49 PM	42	639.36
Test 5	15-Oct	11:35 AM	42	639.36
Avg			42.2	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	1:49 AM	61	1278.72
Test 2	5-Oct	3:28 PM	59	1278.72
Test 3	6-Oct	10:21 AM	60	1278.72
Test 4	9-Oct	1:48 PM	61	1278.72
Test 5	15-Oct	11:36 AM	57	1278.72
Avg			59.6	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	1:48 AM	76	2557.44
Test 2	5-Oct	3:27 PM	73	2557.44
Test 3	6-Oct	10:18 AM	74	2557.44
Test 4	9-Oct	1:46 PM	77	2557.44
Test 5	15-Oct	11:37 AM	69	2557.44
Avg			73.8	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	1:46 AM	121	5115
Test 2	5-Oct	3:25 PM	126	5115
Test 3	6-Oct	10:16 AM	121	5115
Test 4	9-Oct	1:44 PM	136	5115
Test 5	15-Oct	11:39 AM	131	5115
Avg			127	

Appendix K – MapReduce Average Packet Length by Source IP Test Data

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:00:00 AM	17	19.98
Test 2	5-Oct	3:38:00 PM	17	19.98
Test 3	6-Oct	10:33:00 AM	16	19.98
Test 4	9-Oct	1:59:00 PM	17	19.98
Test 5	15-Oct	11:45:00 PM	15	19.98
Avg			16.4	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:00:00 AM	17	39.96
Test 2	5-Oct	3:37:00 AM	17	39.96
Test 3	6-Oct	10:32:00 AM	18	39.96
Test 4	9-Oct	1:59:00 PM	17	39.96
Test 5	15-Oct	11:46:00 PM	16	39.96
Avg			17	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:00:00 AM	20	79.92
Test 2	5-Oct	3:37:00 AM	20	79.92
Test 3	6-Oct	10:32:00 AM	19	79.92
Test 4	9-Oct	1:59:00 PM	20	79.92
Test 5	15-Oct	11:46:00 PM	18	79.92
Avg			19.4	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	1:59:00 AM	22	159.84
Test 2	5-Oct	3:37:00 AM	23	159.84
Test 3	6-Oct	10:31:00 AM	22	159.84
Test 4	9-Oct	1:58:00 PM	22	159.84
Test 5	15-Oct	11:47:00 PM	20	159.84
Avg			21.8	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	1:59:00 AM	29	319.68
Test 2	5-Oct	3:36:00 PM	27	319.68
Test 3	6-Oct	10:30:00 AM	28	319.68
Test 4	9-Oct	1:58:00 PM	29	319.68
Test 5	15-Oct	11:47:00 PM	26	319.68
Avg			27.8	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	1:58:00 AM	43	639.36
Test 2	5-Oct	3:36:00 PM	42	639.36
Test 3	6-Oct	10:30:00 AM	45	639.36
Test 4	9-Oct	1:58:00 PM	48	639.36
Test 5	15-Oct	11:48:00 PM	42	639.36
Avg			44	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	1:56:00 AM	65	1278.72
Test 2	5-Oct	3:34:00 PM	70	1278.72
Test 3	6-Oct	10:28:00 AM	73	1278.72
Test 4	9-Oct	1:56:00 PM	64	1278.72
Test 5	15-Oct	11:50:00 PM	64	1278.72
Avg			67.2	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	1:54:00 AM	81	2557.44
Test 2	5-Oct	3:32:00 PM	87	2557.44
Test 3	6-Oct	10:26:00 AM	90	2557.44
Test 4	9-Oct	1:54:00 AM	89	2557.44
Test 5	15-Oct	11:52:00 PM	91	2557.44
Avg			87.6	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	1:52 AM	154	5115
Test 2	5-Oct	3:31 PM	144	5115
Test 3	6-Oct	10:24:00 AM	133	5115
Test 4	9-Oct	1:52:00 AM	152	5115
Test 5	15-Oct	11:53:00 AM	138	5115
Avg			144.2	

Appendix L – MapReduce Percent of Traffic by Source IP Test Data

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:08 AM	17	19.98
Test 2	5-Oct	3:46 PM	16	19.98
Test 3	6-Oct	10:40 AM	18	19.98
Test 4	9-Oct	2:08 PM	16	19.98
Test 5	15-Oct	11:57 PM	14	19.98
Avg			16.2	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:08 AM	18	39.96
Test 2	5-Oct	3:46 PM	18	39.96
Test 3	6-Oct	10:40 AM	18	39.96
Test 4	9-Oct	2:08 PM	18	39.96
Test 5	15-Oct	11:57 PM	16	39.96
Avg			17.6	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:08 AM	19	79.92
Test 2	5-Oct	3:45 PM	19	79.92
Test 3	6-Oct	10:39 AM	20	79.92
Test 4	9-Oct	2:07 PM	20	79.92
Test 5	15-Oct	11:58 PM	18	79.92
Avg			19.2	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:07 AM	22	159.84
Test 2	5-Oct	3:45 PM	22	159.84
Test 3	6-Oct	10:39 AM	22	159.84
Test 4	9-Oct	2:07 PM	22	159.84
Test 5	15-Oct	1:58 PM	20	159.84
Avg			21.6	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:07 AM	28	319.68
Test 2	5-Oct	3:45 PM	28	319.68
Test 3	6-Oct	10:39 AM	28	319.68
Test 4	9-Oct	2:06 PM	29	319.68
Test 5	15-Oct	11:59 PM	25	319.68
Avg			27.6	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:06 AM	57	639.36
Test 2	5-Oct	3:44 PM	45	639.36
Test 3	6-Oct	10:38 AM	48	639.36
Test 4	9-Oct	2:06 PM	53	639.36
Test 5	15-Oct	12:00 AM	44	639.36
Avg			49.4	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:05 AM	63	1278.72
Test 2	5-Oct	3:43 PM	61	1278.72
Test 3	6-Oct	10:38 AM	68	1278.72
Test 4	9-Oct	12:05 PM	62	1278.72
Test 5	15-Oct	12:00 AM	58	1278.72
Avg			62.4	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:03 AM	80	2557.44
Test 2	5-Oct	3:40 PM	81	2557.44
Test 3	6-Oct	10:35 AM	81	2557.44
Test 4	9-Oct	12:02 PM	96	2557.44
Test 5	15-Oct	12:01 PM	91	2557.44
Avg			85.8	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:00 AM	156	5115
Test 2	5-Oct	3:38 PM	146	5115
Test 3	6-Oct	10:33 AM	139	5115
Test 4	9-Oct	2:00 PM	148	5115
Test 5	15-Oct	12:04 PM	144	5115
Avg			146.6	

Appendix M – Java TCP Port by Source IP Test Data

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:40:00 AM	23.204	19.98
Test 2	5-Oct	4:14:00 PM	22.784	19.98
Test 3	6-Oct	11:45:00 AM	23.01	19.98
Test 4	9-Oct	3:48:00 PM	22.984	19.98
Test 5	15-Oct	12:51:00 AM	23.024	19.98
Avg			23.0012	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	241 AM	45.744	39.96
Test 2	5-Oct	4:15:00 PM	45.363	39.96
Test 3	6-Oct	11:47:00 AM	45.508	39.96
Test 4	9-Oct	3:29:00 PM	45.178	39.96
Test 5	15-Oct	12:52:00 AM	45.497	39.96
Avg			45.458	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:42:00 AM	90.903	79.92
Test 2	5-Oct	4:17:00 PM	90.301	79.92
Test 3	6-Oct	11:50:00 AM	91.103	79.92
Test 4	9-Oct	3:51:00 PM	90.299	79.92
Test 5	15-Oct	12:53:00 AM	90.588	79.92
Avg			90.6388	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:43:00 AM	180.74	159.84
Test 2	5-Oct	4:19:00 PM	178.6	159.84
Test 3	6-Oct	11:55:00 AM	181.36	159.84
Test 4	9-Oct	3:55:00 PM	180.601	159.84
Test 5	15-Oct	12:54:00 PM	180.969	159.84
Avg			180.454	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	3:00:00 AM	361.5	319.68
Test 2	5-Oct	4:28:00 PM	360.28	319.68
Test 3	6-Oct	12:05:00 PM	361.2	319.68
Test 4	9-Oct	4:12:00 PM	361.916	319.68
Test 5	15-Oct	12:58:00 AM	360.468	319.68
Avg			361.0728	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	3:07:00 AM	730.87	639.36
Test 2	5-Oct	4:36:00 PM	720.04	639.36
Test 3	6-Oct	12:18:00 PM	724.5	639.36
Test 4	9-Oct	4:30:00 PM	720.865	639.36
Test 5	15-Oct	1:04:00 AM	722.428	639.36
Avg			723.7406	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	3:32:00 AM	1450.642	1278.72
Test 2	5-Oct	4:02:00 PM	1452.707	1278.72
Test 3	6-Oct	12:31:00 PM	1453.184	1278.72
Test 4	9-Oct	4:45:00 PM	1443.568	1278.72
Test 5	15-Oct	1:16:00 AM	1445.482	1278.72
Avg			1449.1166	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	4:03:00 AM	2918.271	2557.44
Test 2	5-Oct	5:35:00 PM	2920.347	2557.44
Test 3	6-Oct	12:56:00 PM	2919.759	2557.44
Test 4	9-Oct	5:16:00 PM	2920.287	2557.44
Test 5	15-Oct	1:40:00 AM	2911.392	2557.44
Avg			2918.0112	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	4:57:00 AM	9759.133	5115
Test 2	5-Oct	6:27:00 PM	9757.774	5115
Test 3	6-Oct	1:42:00 PM	9755.927	5115
Test 4	9-Oct	6:06:00 PM	9758.349	5115
Test 5	15-Oct	2:33:00 AM	9757.807	5115
Avg			9757.798	

Appendix N – Java Total Traffic by IP Protocol Test Data

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:37:00 AM	0.577	19.98
Test 2	5-Oct	4:06:00 PM	0.4995	19.98
Test 3	6-Oct	11:37:00 AM	0.498	19.98
Test 4	9-Oct	3:41:00 PM	0.567	19.98
Test 5	15-Oct	12:20:00 AM	0.475	19.98
Avg			0.5233	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:37:00 AM	0.937	39.96
Test 2	5-Oct	4:06:00 PM	0.783	39.96
Test 3	6-Oct	11:37:00 AM	0.796	39.96
Test 4	9-Oct	3:41:00 PM	0.908	39.96
Test 5	15-Oct	12:20:00 AM	0.923	39.96
Avg			0.8694	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:37:00 AM	1.569	79.92
Test 2	5-Oct	4:06:00 PM	1.261	79.92
Test 3	6-Oct	11:37:00 AM	1.238	79.92
Test 4	9-Oct	3:41:00 PM	1.51	79.92
Test 5	15-Oct	12:20:00 AM	1.665	79.92
Avg			1.4486	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:37:00 AM	2.783	159.84
Test 2	5-Oct	4:07:00 PM	2.246	159.84
Test 3	6-Oct	11:38:00 AM	2.202	159.84
Test 4	9-Oct	3:42:00 PM	2.91	159.84
Test 5	15-Oct	12:21:00 AM	2.902	159.84
Avg			2.6086	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:37:00 AM	4.351	319.68
Test 2	5-Oct	4:07:00 PM	4.218	319.68
Test 3	6-Oct	11:38:00 AM	4.085	319.68
Test 4	9-Oct	3:42:00 PM	6.355	319.68
Test 5	15-Oct	12:21:00 AM	6.305	319.68
Avg			5.0628	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:37:00 AM	7.895	639.36
Test 2	5-Oct	4:07:00 PM	7.905	639.36
Test 3	6-Oct	11:38:00 AM	7.988	639.36
Test 4	9-Oct	3:42:00 PM	7.843	639.36
Test 5	15-Oct	12:21:00 AM	8.161	639.36
Avg			7.9584	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:38:00 AM	20.146	1278.72
Test 2	5-Oct	4:07:00 PM	16.358	1278.72
Test 3	6-Oct	11:38:00 AM	23.304	1278.72
Test 4	9-Oct	3:42:00 PM	17.609	1278.72
Test 5	15-Oct	12:22:00 AM	16.634	1278.72
Avg			18.8102	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:38:00 AM	83.733	2557.44
Test 2	5-Oct	4:08:00 PM	83.653	2557.44
Test 3	6-Oct	11:39:00 AM	83.738	2557.44
Test 4	9-Oct	3:43:00 PM	83.643	2557.44
Test 5	15-Oct	12:23:00 AM	82.932	2557.44
Avg			83.5398	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:38:00 AM	166.73	5115
Test 2	5-Oct	4:10:00 PM	166.75	5115
Test 3	6-Oct	11:39:00 AM	166.714	5115
Test 4	9-Oct	3:53:00 PM	166.733	5115
Test 5	15-Oct	12:25:00 AM	166.719	5115
Avg			166.7292	

Appendix O – Java Average Packet Length by Source IP Test Data

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:38:00 AM	0.937	19.98
Test 2	5-Oct	4:11:00 AM	0.931	19.98
Test 3	6-Oct	11:39:00 AM	0.897	19.98
Test 4	9-Oct	3:55:00 PM	0.934	19.98
Test 5	15-Oct	12:33:00 AM	0.957	19.98
Avg			0.9312	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:38:00 AM	1.542	39.96
Test 2	5-Oct	4:12:00 AM	1.577	39.96
Test 3	6-Oct	11:40:00 AM	1.621	39.96
Test 4	9-Oct	3:55:00 PM	1.533	39.96
Test 5	15-Oct	12:33:00 AM	1.695	39.96
Avg			1.5936	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:38:00 AM	2.843	79.92
Test 2	5-Oct	4:07:00 PM	3.29	79.92
Test 3	6-Oct	11:40:00 AM	3.347	79.92
Test 4	9-Oct	3:55:00 PM	3.247	79.92
Test 5	15-Oct	12:33:00 AM	3.266	79.92
Avg			3.1986	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:38:00 AM	6.146	159.84
Test 2	5-Oct	4:07:00 PM	6.279	159.84
Test 3	6-Oct	11:40:00 AM	6.117	159.84
Test 4	9-Oct	3:55:00 PM	5.978	159.84
Test 5	15-Oct	12:33:00 AM	6.287	159.84
Avg			6.1614	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:39:00 AM	13.102	319.68
Test 2	5-Oct	4:08:00 PM	10.782	319.68
Test 3	6-Oct	11:40:00 AM	12.352	319.68
Test 4	9-Oct	3:55:00 PM	12.134	319.68
Test 5	15-Oct	12:34:00 AM	13.023	319.68
Avg			12.2786	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:39:00 AM	22.933	639.36
Test 2	5-Oct	4:08:00 PM	21.59	639.36
Test 3	6-Oct	11:41:00 AM	21.641	639.36
Test 4	9-Oct	3:56:00 PM	21.667	639.36
Test 5	15-Oct	12:34:00 AM	21.613	639.36
Avg			21.8888	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:40:00 AM	52.466	1278.72
Test 2	5-Oct	4:09:00 PM	47.396	1278.72
Test 3	6-Oct	11:41:00 AM	44.108	1278.72
Test 4	9-Oct	3:45:00 PM	47.672	1278.72
Test 5	15-Oct	12:35:00 AM	48.053	1278.72
Avg			47.939	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:42:00 AM	112.6	2557.44
Test 2	5-Oct	4:10:00 PM	112.696	2557.44
Test 3	6-Oct	11:42:00 AM	112.589	2557.44
Test 4	9-Oct	3:46:00 PM	113.664	2557.44
Test 5	15-Oct	12:37: AM	112.031	2557.44
Avg			112.716	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:45:00 AM	215.907	5115
Test 2	5-Oct	4:12:00 PM	217.833	5115
Test 3	6-Oct	11:43:00 AM	217.734	5115
Test 4	9-Oct	3:47:00 PM	219.117	5115
Test 5	15-Oct	12:40 AM	225.87	5115
Avg			219.2922	

Appendix P – Java Percent of Traffic by Source IP Test Data

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:34:00 AM	0.967	19.98
Test 2	5-Oct	4:02:00 PM	1.012	19.98
Test 3	6-Oct	11:31:00 AM	0.996	19.98
Test 4	9-Oct	3:36:00 PM	0.905	19.98
Test 5	15-Oct	12:09:00 AM	1.211	19.98
Avg			1.0182	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:35:00 AM	1.708	39.96
Test 2	5-Oct	4:02:00 PM	1.762	39.96
Test 3	6-Oct	11:31:00 AM	1.759	39.96
Test 4	9-Oct	3:36:00 PM	1.655	39.96
Test 5	15-Oct	12:09:00 AM	1.814	39.96
Avg			1.7396	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:35:00 AM	3.094	79.92
Test 2	5-Oct	4:02:00 PM	3.258	79.92
Test 3	6-Oct	11:31:00 AM	3.268	79.92
Test 4	9-Oct	3:36:00 PM	3.148	79.92
Test 5	15-Oct	12:09:00 AM	3.449	79.92
Avg			3.2434	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:35:00 AM	5.898	159.84
Test 2	5-Oct	4:02:00 PM	6.292	159.84
Test 3	6-Oct	11:31:00 AM	6.363	159.84
Test 4	9-Oct	3:36:00 PM	6.178	159.84
Test 5	15-Oct	12:09:00 AM	6.479	159.84
Avg			6.242	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:35:00 AM	13.538	319.68
Test 2	5-Oct	4:02:00 PM	13.441	319.68
Test 3	6-Oct	11:31:00 AM	13.622	319.68
Test 4	9-Oct	3:36:00 PM	14.024	319.68
Test 5	15-Oct	12:09:00 AM	13.441	319.68
Avg			13.6132	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:35:00 AM	28.258	639.36
Test 2	5-Oct	4:03:00 PM	27.728	639.36
Test 3	6-Oct	11:32:00 AM	27.833	639.36
Test 4	9-Oct	3:36:00 PM	28.178	639.36
Test 5	15-Oct	12:09:00 AM	27.709	639.36
Avg			27.9412	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:35:00 AM	53.125	1278.72
Test 2	5-Oct	4:03:00 PM	56.593	1278.72
Test 3	6-Oct	11:32:00 AM	56.345	1278.72
Test 4	9-Oct	3:37:00 PM	56.122	1278.72
Test 5	15-Oct	12:10:00 AM	56.906	1278.72
Avg			55.8182	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:36:00 AM	116.431	2557.44
Test 2	5-Oct	4:04:00 PM	115.773	2557.44
Test 3	6-Oct	11:33:00 AM	115.258	2557.44
Test 4	9-Oct	3:37:00 PM	117.741	2557.44
Test 5	15-Oct	12:11:00 AM	115.419	2557.44
Avg			116.1244	

	Date	Start Time	CPU Time	File Size
Test 1	4-Oct	2:37:00 AM	230.477	5115
Test 2	5-Oct	4:06:00 AM	231.566	5115
Test 3	6-Oct	11:36:00 AM	232.477	5115
Test 4	9-Oct	3:38:00 PM	230.944	5115
Test 5	15-Oct	12:13:00 AM	231.111	5115
Avg			231.315	

Appendix Q: Java Output Samples (All for 20MB File)

IP Address/Source Port Totals

199.17.18.73 57350 9
199.17.18.73 11262 1448
199.17.18.73 5612 11
199.17.18.73 34113 11
107.22.186.115 443 10
Total elapsed time in milliseconds: 23204

Protocol Traffic totals

Protocol = 6 Total = 179897257 bytes
Protocol = 17 Total = 135815 bytes
Total elapsed time in milliseconds: 515

Average Packet Length by Source IP

173.194.66.95 Total = 16361 Avg = 495
52.72.55.108 Total = 2441 Avg = 162
52.6.141.131 Total = 4539 Avg = 453
52.84.14.210 Total = 2002 Avg = 333
104.113.52.120 Total = 50801 Avg = 1154
Total elapsed time in milliseconds: 937

Total Percentage of Traffic by IP

173.194.66.95 0.0091
52.72.55.108 0.0014
52.6.141.131 0.0025
52.84.14.210 0.0011
104.113.52.120 0.0282
Total elapsed time in milliseconds: 967

Appendix R: MapReduce Output Samples (All for 20MB File)

IP Address/Source Port Totals

Protocol = 6 SRC_IP = 98.138.4.113 Port = 443 52
Protocol = 6 SRC_IP = 98.138.49.44 Port = 80 4
Protocol = 6 SRC_IP = 98.138.81.72 Port = 443 19
Protocol = 6 SRC_IP = 98.139.199.205 Port = 443 25

Protocol Traffic totals

17 135815
6 179897257

Average Packet Length by Source IP

98.137.201.232 534
98.138.243.53 534
98.138.4.113 510
98.138.49.44 155
98.138.81.72 451
98.139.199.205 402

Total Percentage of Traffic by IP

199.17.18.73 57350 9
199.17.18.73 11262 1448
199.17.18.73 5612 11
199.17.18.73 34113 11
107.22.186.115 443 10
Total elapsed time in milliseconds: 23204

Appendix S: Cost Analysis Charts

Table S1

Time to Process 50TB/Hour

	MR TCP Port by Source	MR Traffic by IP Protocol	MR Average Packet Length	MR Percent of Traffic	Java TCP Port by Source	Java Traffic by IP Proto	Java Avg Packet length	Java Percent of Traffic
20 MB	12,537.19	11,662.51	11,954.07	11,808.29	16,765.73	381.44	678.76	742.17
40 MB	6,633.05	6,341.49	6,195.71	6,414.38	16,567.32	316.86	580.79	634.00
80 MB	3,425.86	3,316.53	3,535.20	3,498.75	16,516.81	263.97	582.87	591.03
160 MB	1,949.83	1,876.93	1,986.27	1,968.05	16,441.77	237.68	561.39	568.73
320 MB	1,275.59	1,211.81	1,266.48	1,257.36	16,449.27	230.64	559.37	620.17
640 MB	1,102.47	961.25	1,002.25	1,125.25	16,485.60	181.28	535.04	636.45
1.3 GB	758.52	678.79	765.35	710.68	16,504.23	214.23	545.99	635.72
2.6 GB	613.88	420.26	498.85	488.60	16,616.86	475.72	641.87	661.28
5.1 GB	587.11	361.61	410.58	417.41	27,783.30	474.73	624.39	658.62

Table S2

Time to Process 50TB/Day

	MR TCP Port by Source	MR Traffic by IP Protocol	MR Average Packet Length	MR Percent of Traffic	Java TCP Port by Source	Java Traffic by IP Protocol	Java Average Packet length	Java Percent of Traffic
20 MB	522.38	485.94	498.09	492.01	698.57	15.89	28.28	30.92
40 MB	276.38	264.23	258.15	267.27	690.30	13.20	24.20	26.42
80 MB	142.74	138.19	147.30	145.78	688.20	11.00	24.29	24.63
160 MB	81.24	78.21	82.76	82.00	685.07	9.90	23.39	23.70
320 MB	53.15	50.49	52.77	52.39	685.39	9.61	23.31	25.84
640MB	45.94	40.05	41.76	46.89	686.90	7.55	22.29	26.52
1.3GB	31.60	28.28	31.89	29.61	687.68	8.93	22.75	26.49
2.6 GB	25.58	17.51	20.79	20.36	692.37	19.82	26.74	27.55
5.1 GB	24.46	15.07	17.11	17.39	1,157.64	19.78	26.02	27.44

Table S3

CPU Cost to Process 50TB/Hour

	MR TCP Port by Source	MR Traffic by IP Protocol	MR Average Packet Length	MR Percent of Traffic	Java TCP Port by Source	Java Traffic by IP Protocol	Java Average Packet length	Java Percent of Traffic
20 MB	6017.85	3183.86	1644.41	935.92	612.28	529.19	364.09	294.66
40 MB	5598.00	3043.91	1591.93	900.93	581.67	461.40	325.82	201.72
80 MB	5737.95	2973.94	1696.89	953.41	607.91	481.08	367.37	239.45
160 MB	5667.98	3078.90	1679.40	944.66	603.53	540.12	341.13	234.53
320 MB	2011.89	1988.08	1982.02	1973.01	1973.91	1978.27	1980.51	1994.02
640 MB	45.77	38.02	31.68	28.52	27.68	21.75	25.71	57.09
1.3 GB	81.45	69.70	69.94	67.37	67.12	64.20	65.52	77.02
2.6 GB	89.06	76.08	70.92	68.25	74.42	76.37	76.29	79.35
5.1 GB	6017.85	3183.86	1644.41	935.92	612.28	529.19	364.09	294.66

Table S4

Storage Cost to Process 50TB/Hour

	MR TCP Port by Source	MR Traffic by IP Protocol	MR Average Packet Length	MR Percent of Traffic	Java TCP Port by Source	Java Traffic by IP Protocol	Java Average Packet length	Java Percent of Traffic
20 MB	84,712.26	44,818.69	23,148.12	13,174.73	8,618.98	7,449.26	5,125.21	4,147.88
40 MB	78,802.10	42,848.64	22,409.35	12,682.21	8,188.03	6,495.02	4,586.53	2,839.65
80 MB	80,772.15	41,863.62	23,886.89	13,420.98	8,557.42	6,772.06	5,171.39	3,370.64
160 MB	79,787.13	43,341.15	23,640.63	13,297.85	8,495.85	7,603.17	4,802.00	3,301.38
320 MB	37,761.84	37,314.96	37,201.19	37,032.17	37,049.08	37,130.91	37,172.87	37,426.54
640MB	859.12	713.66	594.55	535.33	519.49	408.30	482.52	1,071.49
1.3GB	1,528.78	1,308.13	1,312.81	1,264.42	1,259.89	1,205.07	1,229.74	1,445.70
2.6 GB	1,671.61	1,427.98	1,331.20	1,280.96	1,396.83	1,433.50	1,431.85	1,489.42
5.1 GB	84,712.26	44,818.69	23,148.12	13,174.73	8,618.98	7,449.26	5,125.21	4,147.88

Table S5

Total Cost to Process 50TB/Hour

	MR TCP Port by Source	MR Traffic by IP Protocol	MR Average Packet Length	MR Percent of Traffic	Java TCP Port by Source	Java Traffic by IP Protocol	Java Average Packet length	Java Percent of Traffic
20 MB	90730.11	48002.56	24792.53	14110.64	9231.26	7978.45	5489.30	4442.54
40 MB	84400.10	45892.56	24001.28	13583.14	8769.70	6956.41	4912.35	3041.37
80 MB	86510.11	44837.55	25583.78	14374.39	9165.32	7253.13	5538.76	3610.08
160 MB	85455.10	46420.06	25320.03	14242.52	9099.39	8143.29	5143.13	3535.90
320 MB	39773.73	39303.04	39183.21	39005.19	39023.00	39109.19	39153.37	39420.56
640MB	904.89	751.68	626.23	563.85	547.16	430.05	508.23	1128.57
1.3GB	1610.23	1377.83	1382.76	1331.79	1327.01	1269.28	1295.25	1522.72
2.6 GB	1760.67	1504.06	1402.12	1349.21	1471.25	1509.87	1508.14	1568.77
5.1 GB	90730.11	48002.56	24792.53	14110.64	9231.26	7978.45	5489.30	4442.54