NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

ADDING BIG DATA ANALYTICS TO GCSS-MC

by

Nicholas Bitto

September 2014

Thesis Co-Advisors: Arijit Das
Gurminder Singh

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	09-30-2014	Master's Thesis 03-25-2013 to 09-30-2014

4. TITLE AND SUBTITLE

ADDING BIG DATA ANALYTICS TO GCSS-MC

5. FUNDING NUMBERS

6. AUTHOR(S)

Nicholas Bitto

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Global Combat Support System - Marine Corp is a large logistics system designed to replace numerous legacy systems used by the Marine Corps. While it has been in existence for a while, its intended potential has not been fully realized. Therefore, various teams are working hard to develop the analytics that will benefit the community. With the growth of data, the only way these analytics (in Structured Query Language [SQL]) will run efficiently will be on proprietary hardware from Oracle. This research looks at running the same analytics on commodity hardware using Hadoop Distributed File System and Java Map Reduce. The results show that while it takes longer to program in Java (over SQL), the analytics are just as, or even more powerful ,as SQL, and the potential to save on hardware cost is significant.

14. SUBJECT TERMS

Big Data, Hadoop, MapReduce, GCSS-MC

15. NUMBER OF PAGES 93

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

ADDING BIG DATA ANALYTICS TO GCSS-MC

Nicholas Bitto
Lieutenant, United States Navy
B.S. , Old Dominion University, 2009

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2014

Author: Nicholas Bitto

Approved by: Arijit Das
Thesis Co-Advisor

Gurminder Singh Ph.D.
Thesis Co-Advisor

Peter J. Denning Ph.D.
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Global Combat Support System - Marine Corp is a large logistics system designed to replace numerous legacy systems used by the Marine Corps. While it has been in existence for a while, its intended potential has not been fully realized. Therefore, various teams are working hard to develop the analytics that will benefit the community. With the growth of data, the only way these analytics (in Structured Query Language [SQL]) will run efficiently will be on proprietary hardware from Oracle. This research looks at running the same analytics on commodity hardware using Hadoop Distributed File System and Java Map Reduce. The results show that while it takes longer to program in Java (over SQL), the analytics are just as, or even more powerful ,as SQL, and the potential to save on hardware cost is significant.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

1 Introduction **1**

1.1 The Problem 1

1.2 Research Questions 2

1.3 Contributions 2

1.4 Thesis Outline 2

2 The Current State **5**

2.1 GCSS-MC: The USMC ERP Solution 6

2.2 Big Data 7

2.3 MapReduce 8

2.4 Word Count 10

2.5 Hadoop 19

3 The Experiment Design **23**

3.1 Why Add a Big Data Element? 23

3.2 Adding a Big Data Element to GCSS-MC 24

3.3 Building a Hadoop Cluster 25

4 The Experiment **29**

4.1 The GCSS-MC Data 29

4.2 Top-100-NSN Program 32

4.3 SQL Simulation 45

5 Conclusion **57**

5.1 The Outcome 57

5.2 Future Work 59

Appendices

A Hadoop Testing on Single and Two Node Clusters **61**

A.1 Hypothesis . . . 62

A.2 Experiment Architecture . . . 62

A.3 Experiments Run . . . 63

A.4 Results . . . 64

A.5 Conclusion . . . 65

B How to Setup a Single Node Hadoop Cluster **67**

References **73**

Initial Distribution List **75**

List of Figures

Figure 2.1 GCSS-MC Architecture 7
Figure 2.2 Google Execution Overview 10
Figure 2.3 Hadoop WordCount Program 11
Figure 2.4 Hadoop WordCount Program: Libraries 12
Figure 2.5 Hadoop WordCount Program: Map Class 13
Figure 2.6 Hadoop WordCount Program: Reduce Class 14
Figure 2.7 Hadoop WordCount Program: Main Method 15
Figure 2.8 Jack and Jill Nursery Rhyme 16
Figure 2.9 Byte Offset for Jack and Jill Nursery Rhyme 16
Figure 2.10 Intermediate Key/Value Pairs from the Map Method 17
Figure 2.11 Shuffled Intermediate Key/Value Pairs from the Map Method . . 18
Figure 2.12 Final Output of WordCount 19
Figure 2.13 HDFS Architecture 20
Figure 2.14 HDFS JobTracker Interaction 22

Figure 3.1 Experiment Architecture 24

Figure 4.1 Sample GCSS-MC data in JSON Format 30
Figure 4.2 Data Flow through the NSN Program 34
Figure 4.3 NSN: First Algorithm 35
Figure 4.4 NSN Output from the First Algorithm 36
Figure 4.5 NSN: Second Algorithm 37
Figure 4.6 Descending Order Sort Class 38

Figure 4.7 NSN Output from the Second Algorithm 38
Figure 4.8 NSN: Third Algorithm 39
Figure 4.9 NSN Output from the Third Algorithm 40
Figure 4.10 NSN: Fourth Algorithm 41
Figure 4.11 NSN Output from the Fourth Algorithm 41
Figure 4.12 NSN: Fifth Algorithm 43
Figure 4.13 NSN Output from the Fifth Algorithm 44
Figure 4.14 Alpha Program Logic 45
Figure 4.15 Data Flow Through the Alpha Program 46
Figure 4.16 Alpha: First Algorithm 48
Figure 4.17 Alpha Output from the First Algorithm 49
Figure 4.18 Alpha: Second Algorithm 50
Figure 4.19 Alpha Output from the Second Algorithm 51
Figure 4.25 Alpha Output from the Fifth Algorithm 51
Figure 4.20 Alpha: Third Algorithm 52
Figure 4.21 Alpha Output from the Third Algorithm 53
Figure 4.22 Alpha: Fourth Algorithm 54
Figure 4.23 Alpha Output from the Fourth Algorithm 54
Figure 4.24 Alpha: Fifth Algorithm 55

Figure A.1 Experiment Architecture 63
Figure A.2 Write Benchmark Results 64
Figure A.3 Read Benchmark Results 65
Figure A.4 TeraSort Benchmark Results 65

List of Tables

Table 3.1 Experiment Server Specifications 26

Table 3.2 Experiment HDFS Node Specifications 26

Table 4.1 Tables Written Back to Database 42

Table 4.2 Sample of HDFS_R001_INVENTORY_NSN Table 44

Table 4.3 Sample of HDFS_ALPHA Table 53

Table A.1 Initial Node Settings 62

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

COTS	commercial off the shelf
CSV	Comma Separated Value
DOD	Department of Defense
ERP	enterprise resource planning
GAO	Government Accountability Office
GB	gigabyte
GCSS-MC	Global Combat Support System - Marine Corp
GFS	Google File System
HD	hard drive
HDFS	Hadoop Distributed File System
IDC	International Data Corporation
IP	Internet Protocol
JDBC	Java Database Connectivity
JSON	JavaScript Object Notation
NSN	National Stock Number
PB	petabyte
PC	personnel computer
PL/SQL	Procedural Language/Structured Query Language
RAM	random access memory
SQL	Structured Query Language

TB terabyte

USMC United States Marine Corp

VM virtual machine

YB yottabyte

ZB zettabyte

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisors Arijit Das and Gurminder Singh for the continuous support of my research. Their guidance and patience helped me tremendously during the creation, research, and writing of this thesis. I could not have imagined having better advisors for my thesis.

I would also like to thank my family: my wife, Laura Bitto, for taking care of our daughter during all of the late nights and weekends that went into completing this thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

Big data has become a buzz word in both the government and private sectors. That does not mean that big data is not a worthwhile venture. There is no hiding from the fact that data continues to grow, and processing enormous amounts of data using conventional techniques can become unmanageable. For example, the Pentagon is attempting to expand its worldwide communications network to handle yottabyte (YB)s (10^{24} bytes) of data (A YB one trillion terabyte (TB)s) [1]. As the data increases to new thresholds, current database architectures struggle to keep up. This thesis examines the Global Combat Support System - Marine Corp (GCSS-MC) database and how to apply a big data solution.

1.1 The Problem

The amount of data being stored in database is increasing both in the government and public sectors. The United States Marine Corp (USMC) is no different, and the need to access, store, and process large amounts of data exists. The USMC is currently working on an enterprise resource planning (ERP) system that will replace all of its legacy logistics systems. The solution that has been developed is called GCSS-MC. GCSS-MC is a complex undertaking and has seen its share of problems during development. The system is making significant achievements and is becoming a useful resource to the USMC. However, the design of this system has taken a significant amount of time to implement while the data continues to grow.

Initially, the GCSS-MC did not consider a big data element in the system. But as the amount of data and desire to keep data is increasing, the USMC needs to find ways to add a big data solution to GCSS-MC. This thesis will explore a method to implement a big data element into GCSS-MC, by adding a Hadoop Distributed File System (HDFS) cluster. Furthermore, a method will be discussed to show no hardware changes will need to be made to the existing GCSS-MC architecture.

1.2 Research Questions

The research and work of this thesis will be focused on addressing the following questions:

- What would an architecture look like that adds a big data element to GCSS-MC?
- If an architecture can be developed, what modifications would the GCSS-MC architecture need?
- How can the data contained in GCSS-MC be imported into HDFS?
- What type of analytics can Hadoop provide for GCSS-MC data?
- How will the data get back to the GCSS-MC database?

1.3 Contributions

The GCSS-MC database contains structured data and is stored in a relational database. This data will be accessed and then parsed and stored in the HDFS ecosystem. The data will then be used in HDFS to run analytics for the GCSS-MC system. The interesting idea in that concept is that HDFS will be used to examine all of the data in the corpus and then perform some calculations and return the data in a different format to the GCSS-MC database. This becomes particularly useful when there is only a small amount of data that needs to be derived and displayed from the entire data corpus. This is illustrated in two separate analytic programs. The programs are written to show that any analytic that can be run in a typical database operation can be accomplished in HDFS. Furthermore, HDFS can be used to perform the exact same queries that a Structured Query Language (SQL) statement can perform. This is increasingly interesting when the data exceeds the capability of standard database designs.

1.4 Thesis Outline

The rest of the chapters in this thesis will further expand on the aforementioned ideas and research questions. Below is a brief outline of what the reader can expect to find in each chapter.

Chapter 2 serves as a starting point for the technology covered in this thesis. The chapter reviews the current architecture and status of the GCSS-MC system. The chapter also discusses the start of the MapReduce paradigm through the introduction of Jeffrey Dean and Sanjay Ghemawat's paper on the MapReduce. Furthermore, an explanation and walk

through of the canonical WordCount example is given. The chapter wraps up with an introduction of the HDFS architecture.

The experiment architecture is introduced in Chapter 3. The chapter begins with some statistics to further illustrate the need for big data solutions. The chapter then outlines how a big data element could be added to the current GCSS-MC architecture. Finally, it explains how the Hadoop cluster is created, and some implementation decisions that were required.

Chapter 4 details the specifics of the experiment. First, there is a discussion of the GCSS-MC data and how it was accessed. Then the chapter delineates how the data is parsed and stored in the Hadoop ecosystem. There are also two separate analytics discussed in Chapter 4. The first analytic is the Top 100 National Stock Number (NSN) program which will be shown in detail. The second analytic is a program called Alpha, an example of SQL simulation. Alpha is also discussed and shown in detail.

The final chapter summarizes the findings and reiterates the research questions that are presented in the thesis. Chapter 5 also discusses some of the possible future work. Such as: adding big data tools (Hive, Squoop, Hbase, etc.), running the experiment code on a production level HDFS ecosystem, and further optimizing the code.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
The Current State

The Department of Defense (DOD) creates, monitors, and views petabyte (PB)s of data every day. The current data management software and hardware make it very difficult to manage, organize, and parse large datasets. Moreover, the data resides across several databases. This infrastructure makes it difficult, or even impossible, for a user who needs to aggregate data from several databases to make an intelligent deductions on that data. The DOD has significant work remaining to streamline all of their data into one space. There are several projects working on a solution to place all of the data in a cloud environment that allows all users to access the data payload. These programs vary greatly across the DOD and no front-runner sets the standard. Even before the DOD can start using the cloud as a large data store, several steps need to take place to get the DOD ready to move to a cloud environment.

The DOD took on the challenge of bringing all of the legacy business systems into a cohesive system in the 1990s. DOD's business systems are information systems, including financial and non-financial systems that support DOD business operations, such as civilian personnel, finance, health, logistics, military personnel, procurement, and transportation [2]. This process has become known as the ERP and entails an automated system using commercial off the shelf (COTS) software consisting of multiple, integrated functional modules that perform a variety of business related tasks such as general ledger accounting, payroll, and supply chain management [2]. ERP processes across the DOD have been widely publicized for being behind schedule and cost overruns (because of the size, complexity, and significance to the DOD). ERP has been placed on the Government Accountability Office (GAO) high-risk list [2].

In total, there are nine ERP solutions being developed for the DOD. All of these systems hope to replace over 500 legacy systems. Replacing these systems in to aggregate systems data will save the DOD money and give the end user a more powerful system that they can use to help manage their work and use to run analytics that can possibly help them improve their work methods. The ERP solution is an ongoing process, that recently has

put in significant efforts towards a successful project. The USMC solution for ERP is the GCSS-MC. The GCSS-MC will replace legacy systems and tailor a solution that is useful to both the Marine in the field and the Marine shipping the supplies.

2.1 GCSS-MC: The USMC ERP Solution

The GCSS-MC system was designed to run off of COTS architecture and combine all of the USMC legacy business systems in to one system. This process has been ongoing since November 2003, and it was intended to deliver integrated functionality across the logistics areas. [3] This process has been incremented slowly through the delivery of COTS architecture to bring all of the USMC logistic tools to one place. Figure 2.1 is an example of the system architecture, from [4]. The program has had its setbacks and successes. The overwhelming process of replacing up to forty-eight legacy systems is not easily accomplished [3]. This system is attempting to design a replacement for all legacy systems in one place, which makes the GCSS-MC system incredibly complex. The system has to be designed to meet the needs of the USMC both at the garrison and in the field, all while maintaining real time data updates over the entire system. One can see that this is a difficult undertaking and should not be taken lightly. The point of this thesis is not to take on the challenge of what should or should not have been done either during design or the implementation of GCSS-MC. Moreover, this thesis does not aim to replace and change the current architecture of the GCSS-MC system. The main aim of this thesis is to show a proof of concept that big data analytics can be added to the current GCSS-MC architecture. GCSS-MC uses proven Oracle technology to give the users a reliable relational database. The work on GCSS-MC is not done and its implementation process continues, but does not yet incorporate a big data element. Big Data elements would allow the USMC the ability to store and analyze more data. Currently the only data that GCSS-MC addresses is structured data.

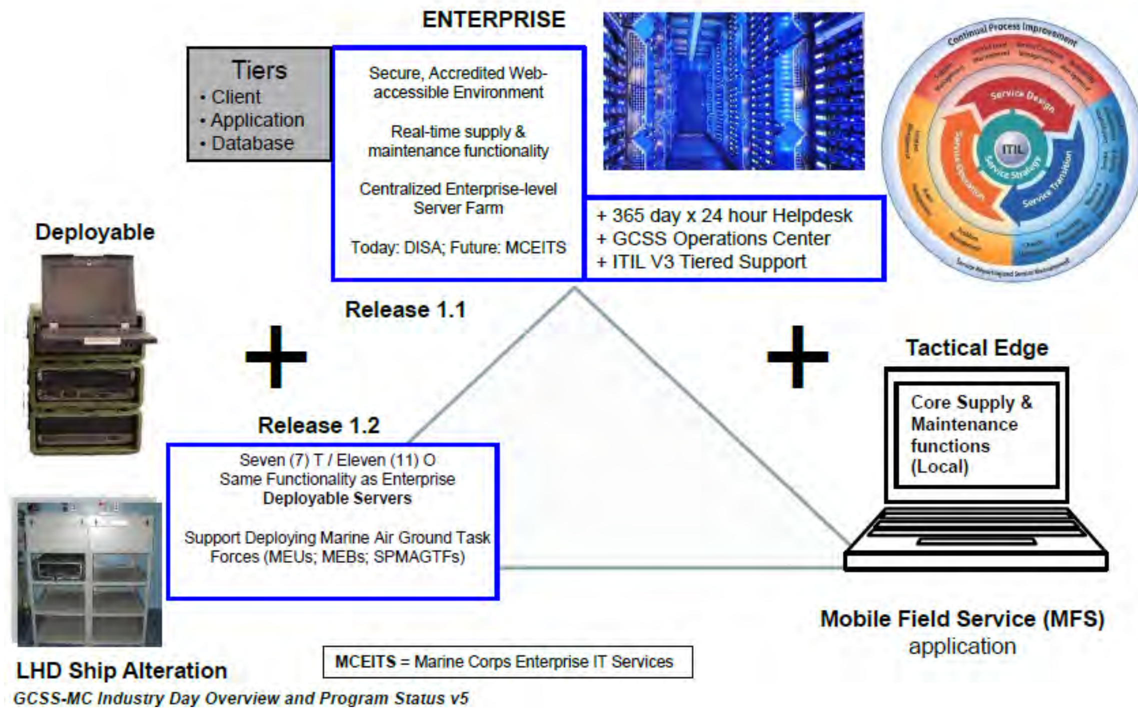


Figure 2.1: GCSS-MC Architecture

2.2 Big Data

Big data has become a buzz word both in industry and the government sector; evidence of this is shown when President Obama started a Big Data Research and Development Initiative in March 2012 [5]. It has helped shift how data is thought about and what can be done with data. Big data does not answer all questions and there are limitations to what it can do. Big data can take the form of MapReduce, NoSQL, Big Table, or something completely different. The Big data industry changes rapidly, and new tools and architecture are constantly being introduced. These tools all provide some benefit and some drawbacks. No one technology provides everything an end user would want. There is currently no defacto standard and the arguments exists as to what technology is the best. The simple answer is that they all provide something, and it really depends on what type of data, how much data, and what the desired analytics are.

2.3 MapReduce

One of the major core technologies for big data is MapReduce. This process is used heavily at Google and in Apache's HDFS system [6], [7]. Google's development in the Map Reduce space is largely what pushed the open source community to answer with HDFS [8]. Google's efforts were documented in Jeffrey Dean and Sanjay Ghemawat's paper on the MapReduce paradigm [6]. This paper is what gave the data science community a whole a look at how Google operated their computer cluster to perform a large number of tasks. The Google MapReduce paper was the first publication illustrating a clustered MapReduce framework. Since this paper has been published several other models have been developed Google's model. Apache Hadoop is an example of cluster computing environment created after this paper was published. This paper is accepted as a seminal paper and has been cited in thousands of other publications.

MapReduce is a programming model that was developed by Google as an answer to how to process and create large amounts of data. At Google, MapReduce jobs are run everyday and they process more than twenty PBs of data per data [6]. The programming model was developed to abstract the complexity of cluster computing away from the programmer. Parallel computing is extremely difficult, and if the programmer has to handle how to parallelize the computation, how the system distributes the data around the cluster, failures of cluster nodes, and as well as the data, the task can be overwhelming. Google developed a system and a programming model that abstracts that from the user. The programmer can then write code that can be executed on the cluster though the Google File System [9]. The program then only has to deal with writing the code to handle the calculation and the distribution of the data; how it is parallelized is all handled by the underlying system. This allows the programmer to focus on the algorithms and not specify the parallel features.

Google found that most of their data computations involved mapping the data to a computation and then storing the resultant key/value pairs. Google further took their inspiration from the map and reduce primitives that were available in Lisp [6]. Furthermore, this led to the development of the MapReduce programming model. The programmer will specify both the map and reduce function to handle the data in key/value pairs. The map function will first take the data and map it to a logical record of intermediate key/value pairs. Then the intermediate key/value pairs are sent to the reduce function to apply the same function

to all of the values that are the same. The canonical example of this is the word count problem that will be further discussed in Section 2.4.

Google focused on implementing this cluster on a large number of commodity personnel computer (PC)s with a Gigabit Ethernet network [9]. This allowed them to quickly and cheaply build a large scale cluster. This idea of using commodity PCs is important because prior to this framework being developed, the conventional thinking was to have a super computing environment which needed very powerful, expensive servers. The fact that Google showed that the computational power, storage, and speed could be mimicked with a bunch of PCs allowed the parallel computing community to expand into other areas and make super computing more achievable to more people [10]. Google's implementation also has thousands of machines, and with that many machines failures are common. Since failures are common, Google's framework supports fault tolerance and allows a machine to be replaced, and since the machines are PCs, the cost to replace them is significantly lower than a server.

The Google framework executes [6] the MapReduce program by first splitting the input files into pieces, these pieces typically range from 16-64MB per piece. These are then sent to worker nodes, and a map function is then started up on each worker node that has been assigned a data split. One of the worker nodes is designated as the master. The master is responsible for designating the worker nodes for both the map and reduce functions. he master will attempt to assign the tasks to idle nodes in the cluster. Each worker that is assigned a data input split will perform a map function on the input data and will parse the input data into intermediate key/value pairs. The values are written to the local disk through a buffer. The location of the intermediate values is then sent to the master. This is needed so the master knows where to tell the reduce workers where their input data resides. A reduce worker node will then make a call to get the data from the location passed to it by the master. The reduce node will read all of the data into its partition, and it will sort the intermediate data be the key value. The reduce node will then iterate over the data for each unique key value and send that to the reduce function specified by the program. The reduce node appends its output to the program specified output file. When all of the map and reduce tasks report completion to the master, the master will wake up the program and the MapReduce call is returned back to that program. A pictorial example of this process

is shown in Figure 2.2, from [6].

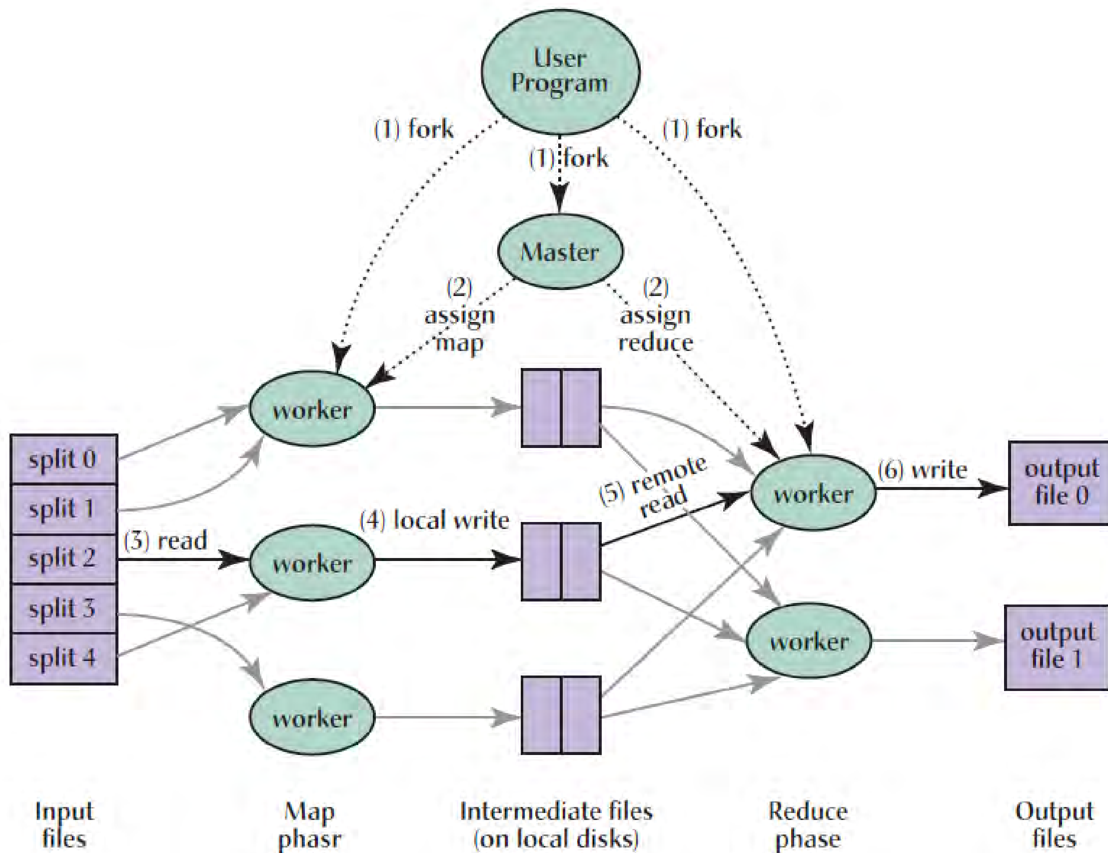


Figure 2.2: Google Execution Overview

2.4 Word Count

The MapReduce programming paradigm can be used to process all kinds of data and can become very convoluted. The widely accepted "Hello World" of MapReduce is the word count program. This example has been used by Google [6] and is also used as an example by Hadoop [11]. Figure 2.3 is the source code for the WordCount program form the Hadoop web page [11]. This is the example code that we will step through to example the MapReduce process.

```java
package org.myorg;

import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

public class WordCount {

   public static class Map extends MapReduceBase implements Mapper<LongWritable, Text,
  Text, IntWritable> {
     private final static IntWritable one = new IntWritable(1);
     private Text word = new Text();

     public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output,
      Reporter reporter) throws IOException {
       String line = value.toString();
       StringTokenizer tokenizer = new StringTokenizer(line);
       while (tokenizer.hasMoreTokens()) {
         word.set(tokenizer.nextToken());
         output.collect(word, one);
       }
     }
  }

   public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable,
  Text, IntWritable>{
     public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text,
  IntWritable> output, Reporter reporter) throws IOException {
       int sum = 0;
       while (values.hasNext()) {
         sum += values.next().get();
       }
       output.collect(key, new IntWritable(sum));
     }
   }

   public static void main(String[] args) throws Exception {
     JobConf conf = new JobConf(WordCount.class);
     conf.setJobName("wordcount");

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

     conf.setMapperClass(Map.class);
     conf.setCombinerClass(Reduce.class);
     conf.setReducerClass(Reduce.class);

     conf.setInputFormat(TextInputFormat.class);
     conf.setOutputFormat(TextOutputFormat.class);

     FileInputFormat.setInputPaths(conf, new Path(args[0]));
     FileOutputFormat.setOutputPath(conf, new Path(args[1]));

     JobClient.runJob(conf);
   }
}
```

Figure 2.3: Hadoop WordCount Program

The first section of code (Figure 2.4) allows the program access to the Hadoop libraries needed to execute the code in the Hadoop environment. The first two packages are Java packages that are not specific to the Hadoop Libraries. The first imported package is the exception package. This allows the program to throw the exceptions it encounters and will provide a graceful shutdown and some help in debugging. The next import is the Java utilities package that allows the program to have access to the iterator class and the string tokenizer class. The next five packages are all Hadoop specific. The org.apache.hadoop.fs.Path package allows the program to access the Hadoop file system. This is needed both for the import of the files needed to run the program and the export path of the result. The next package, org.apache.hadoop.conf.*, is needed to set the job up. This is the part of the code that is executed in the main method. The package org.apache.hadoop.io.* is need to write to the read into the program and write out of the program and is additionally needed to write to logs,stdout, and stderr. The Hadoop utility package, org.apache.hadoop.util.*, is used to report progress on the program during execution. Finally, the org.apache.hadoop.mapred.* package is where the program is getting access to the majority of the classes need. This package includes the class definitions for the map and reduce methods. It will also give the program the class definitions for the input format, the input split, how the job is configured, the output collector, and the output format.

```
1       import java.io.IOException;
2       import java.util.*;
3
4       import org.apache.hadoop.fs.Path;
5       import org.apache.hadoop.conf.*;
6       import org.apache.hadoop.io.*;
7       import org.apache.hadoop.mapred.*;
8       import org.apache.hadoop.util.*;
```

Figure 2.4: Hadoop WordCount Program: Libraries

The map class is going to be the next area of focus (Figure 2.5). The program first has to make the map deceleration.

```
1   public static class Map extends MapReduceBase implements Mapper<LongWritable, Text,
2         Text, IntWritable>
```

In this case, Map is the program defined class name and it extends all of the properties of the MapReduceBase class. The MapReduceBase is what allows the Map class to be called in the job. Next the Map class must implement the Mapper as an interface. This is the class that actually maps the input to the intermediate output. The program must specify the input key/value data type and the output key/value pair data type that will become the intermediate data. Here the input key is the byte offset of the line from the input document and the value is the entire line of the input. The map class then must declare a map method.

```
1  public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output,
2        Reporter reporter)
```

The method will again specify the input key/value pairs. The method will declare the OutputCollector interface and set the key/value output type and give the OutputCollector a name. The Reporter type is what will report status back to the Hadoop system. This map method set the output to be Text and an IntWritable, both of which are Hadoop types. Finally the output statement collects writes the intermediate data out.

```
1  output.collect(word, one);
```

```
1      public static class Map extends MapReduceBase implements Mapper<LongWritable, Text,
2     Text, IntWritable> {
3        private final static IntWritable one = new IntWritable(1);
4        private Text word = new Text();
5
6        public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output,
7         Reporter reporter) throws IOException {
8          String line = value.toString();
9          StringTokenizer tokenizer = new StringTokenizer(line);
10         while (tokenizer.hasMoreTokens()) {
11           word.set(tokenizer.nextToken());
12           output.collect(word, one);
13         }
14       }
15    }
```

Figure 2.5: Hadoop WordCount Program: Map Class

Next the reduce class will take over (Figure 2.6). The reduce class has to be specified in the program and must accept the input data types defined by the map data type output. The reduce class declaration is the first thing that must be considered.

```
1 public static class Reduce extends MapReduceBase implements Reducer<Text,
2         IntWritable, Text, IntWritable>
```

As with the map class, in order to run as a job the reduce class must also extend the MapReduceBase class. Then, the reduce class implements the Reducer interface and must declare the key/value input and output data types. This example shows the case where the reduce class key/value input and output are the same; this is not a constraint. The reduce class could output whatever the programmer chooses as long as it is a Hadoop data type. The reduce class must define a reduce method.

```
1 public void reduce(Text key, Iterator <IntWritable> values,
2         OutputCollector <Text,IntWritable> output, Reporter reporter)
```

The reduce method declaration is similar to the map declaration. The OutputCollector and Reporter declarations are the same. The difference here is the declaration of the input key value pairs. The method must define the key type and then an iterator for the input value type. This is because before the intermediate data gets to the reduce class, it will be shuffled, and the reduce class will have all the the intermediate data that has the same key. So the reduce method will set the key and the iterate over all the values that exist in the intermediate data that have the same key that is set. This is where the output key/value pairs are reduced in the manner specified by the program. In this example, the final value for each key is just the sum of all of the values. Finally, the reduce class has to set the output. This is a similar to call the map output statement.

```
1 output.collect(key, new IntWritable(sum));
```

```
1     public static class Reduce extends MapReduceBase implements Reducer<Text,IntWritable,
2    Text,IntWritable>{
3       public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text,
4    IntWritable> output, Reporter reporter) throws IOException {
5         int sum = 0;
6         while (values.hasNext()) {
7           sum += values.next().get();
8         }
9         output.collect(key, new IntWritable(sum));
10      }
11    }
```

Figure 2.6: Hadoop WordCount Program: Reduce Class

The final piece the program must implement is a main method (Figure 2.7). The main method is responsible for the set up of the MapReduce job. This is done by declaring a JobConf object and telling it where the map and reduce methods are found. This example the classes are found in the WordCount class.

```
1  JobConf conf = new JobConf(WordCount.class);
```

The JobConf typically sets the mapper, reducer, input format, output format [12], but it also allows the programmer to manipulate the job here. The programmer has the access here to set specific combiner classes, use distributed cache, use custom comparators, and even manipulate how the program executes by changing memory requirements for the map and reduce tasks. The JobConf also allows the setting of the input and output path. In this example it is set form the command line arguments.

```
1          JobConf conf = new JobConf(WordCount.class);
2          conf.setJobName("wordcount");
3
4          conf.setOutputKeyClass(Text.class);
5          conf.setOutputValueClass(IntWritable.class);
6
7          conf.setMapperClass(Map.class);
8          conf.setCombinerClass(Reduce.class);
9          conf.setReducerClass(Reduce.class);
10
11         conf.setInputFormat(TextInputFormat.class);
12         conf.setOutputFormat(TextOutputFormat.class);
13
14         FileInputFormat.setInputPaths(conf, new Path(args[0]));
15         FileOutputFormat.setOutputPath(conf, new Path(args[1]));
16
17         JobClient.runJob(conf);
18     }
19  }
```

Figure 2.7: Hadoop WordCount Program: Main Method

The final piece to wrap up here is a walk through of the WordCount program execution. The idea is to use a small data set and walk through the program execution to follow the data flow and data transformation. This example will look at the nursery rhyme Jack and Jill (Figure 2.8).

```
1  Jack and Jill went up the hill
2  To fetch a pail of water.
3  Jack fell down and broke his crown,
4  And Jill came tumbling after
```

Figure 2.8: Jack and Jill Nursery Rhyme

For the purpose of this walk through, we will only consider one map task and one reduce task. The first thing that will happen is the first line of the file will be read in by the Map class. The first line will come in with a byte offset of 0, therefore the key value be zero and the value of the line is "Jack and Jill went up the hill". Figure 2.9 shows the input key/values pairs for the whole text.

```
1  0 Jack and Jill went up the hill
2  31 To fetch a pail of water.
3  57 Jack fell down and broke his crown,
4  93 And Jill came tumbling after
```

Figure 2.9: Byte Offset for Jack and Jill Nursery Rhyme

The line is then parsed using a StringTokenizer, which sets each word as a token. After the tokens are set, a while loop is set up on the condition that there are more tokens. This while loop will then set a Text variable to the String value of each token and then set the output to the word and one. This sends an intermediate data value of ("word", 1) for each word in the line. The intermediate value of the first line will be as follows:

```
1  Jack 1
2  and  1
3  Jill  1
4  went  1
5  up    1
6  the   1
7  hill 1
```

This process is repeated for each line in the input file producing intermediate outputs of ("word", 1) for each word that occurs in the file. This produces the intermediate data that

will then be sorted and shuffled prior to going to the reduce method, Figure 2.10 illustrates the intermediate values for the entire file.

```
1  Jack 1
2  and 1
3  Jill 1
4  went 1
5  up 1
6  the 1
7  hill 1
8  To 1
9  fetch 1
10 a 1
11 pail 1
12 of 1
13 water 1
14 Jack 1
15 fell 1
16 down 1
17 and 1
18 broke 1
19 his 1
20 crown 1
21 And 1
22 Jill 1
23 came 1
24 tumbling 1
25 after 1
```

Figure 2.10: Intermediate Key/Value Pairs from the Map Method

The sorting and shuffling process then takes over and sorts the data. The data is sorted into alphabetical order based on the key. The shuffling process would shuffle the data into logical units and then send data in logical sets to reduce tasks. Again, there is only one reduce task so it will get the entire data set. The input to the reduce method is illustrated in Figure 2.11.

```
1   And 1
2   Jack 1
3   Jack 1
4   Jill 1
5   Jill 1
6   To 1
7   a 1
8   after 1
9   and 1
10  and 1
11  broke 1
12  came 1
13  crown 1
14  down 1
15  fell 1
16  fetch 1
17  hill 1
18  his 1
19  of 1
20  pail 1
21  the 1
22  tumbling 1
23  up 1
24  water 1
25  went 1
```

Figure 2.11: Shuffled Intermediate Key/Value Pairs from the Map Method

Notice this program looks at "and" and "And" as different words. This is how the map method is programmed, and if this result would be undesirable, then programmer could use string manipulation or Regex to get rid of this. The reduce method will then take over. Here, the reduce method will take each key in and iterate over all values for that key. In this reduce method, a int variable is set and used to sum the number of occurrences of each word. For instance, the word "Jack" is seen twice, the reducer will set the value of sum to one on the first occurrence of "Jack" by the call:

```
1   sum += values.next().get();
```

At this point, the sum is set to one. On the next occurrence of "Jack", the same sum call is executed, and then sum is set to two. This process occurs until there are no more values for the key "Jack". In this case, there are no more values, so the reduce class then makes the

call to write the output of the key and the sum by:

```
1 output.collect(key, new IntWritable(sum));
```

The reduce method will get called for each unique key in the intermediate data and write one output statement for each key until is has seen all of the keys in the intermediate data. The final result in this example is seen in Figure 2.12.

```
1  And 1
2  Jack 2
3  Jill 2
4  To 1
5  a 1
6  after 1
7  and 2
8  broke 1
9  came 1
10 crown 1
11 down 1
12 fell 1
13 fetch 1
14 hill 1
15 his 1
16 of 1
17 pail 1
18 the 1
19 tumbling 1
20 up 1
21 water 1
22 went 1
```

Figure 2.12: Final Output of WordCount

2.5 Hadoop

The Hadoop Distributed File System is an open source project that was modeled after the Google Architecture outlined in Jeffery Dean and Sanjay Ghemawat's MapReduce paper [6]. Hadoop started out as an open source web search engine called Nutch [7]. The Nutch project was having some trouble handling the distributed nature of computations that were needed for a web search engine. Then, from the Google papers on MapReduce and the Google File System (GFS), the project began to catch hold [7], [6], [9]. At that time,

Yahoo! became interested in the project, and Hadoop split from the Nutch project and became what it is today. HDFS is the distributed file system that supports MapReduce framework, and Figure 2.13 illustrates the HDFS architecture, from [11]. The distributed file system is set up to be fault tolerant and spread data over several nodes in a cluster. The data replication factor is nominally set to three. However, this can be changed and adapted to meet the specific needs of a given cluster. The most basic cluster, a single node, has the following services running NameNode, JobTracker, TaskTracker, Secondary NameNode, and DataNode.

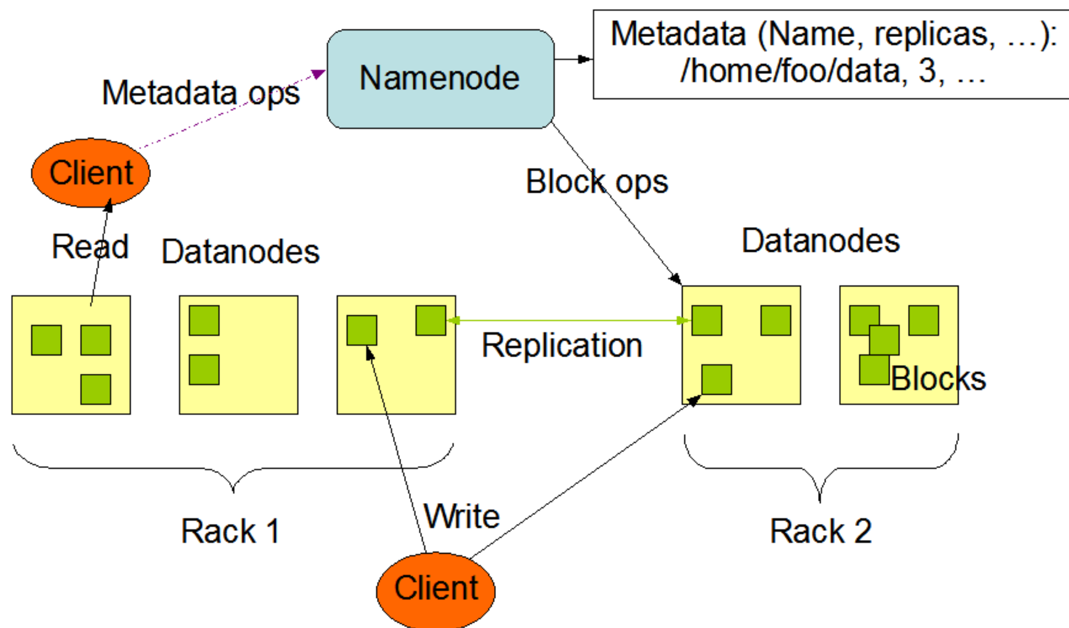


Figure 2.13: HDFS Architecture

The NameNode is known as the master in the cluster. The NameNode is responsible for the file system namespace [13]. The NameNode is the node that manages the whole cluster. It maintains the file system tree and stores all the metadata for all of the directories and files within the distributed file system. The NameNode is the most important node in the cluster because it is needed to maintain all of the files in the distributed file system. If the NameNode were lost than all of the files in the distributed file system would be lost because the NameNode maintains the file structure to be able to put the blocks back together. The

NameNode stores the metadata and file tree in two files; the namespace image and an edit log [7]. Since, HDFS stores all of the files in blocks, the NameNode will keep a reference to every block that exists in the file system.

The Secondary NameNode is there to backup the NameNode. However, this is not a true backup in the sense that if the NameNode fails; the Seocndary NameNode would take over. The main purpose of the Secondary NameNode is to periodically merge the namespace image with edit log. This will prevent the edit log from becoming too large [7]. The Secondary NameNode keeps a copy of the merged edit log and file system image. It is important to note that this is not real time, that is to say that if the HDFS is restored from the Secondary NameNode image, there will be lost data from the time difference of when NameNode failure occurred and the last edit log merge. HDFS does provide a way for the NameNode to write the persistent state to several file systems to reduce the chance of all the file systems failing at once.

The JobTracker is responsible for accepting jobs and then dividing the jobs into tasks and assigning those tasks to DataNodes [14]. The JobTracker will try to do the best it can to maintain data locality, meaning that it will try to give the tasks to the DataNodes that physically have the blocks of data the task is to be executed on. The JobTracker will access the NameNode to be able to determine which nodes physically have the data. This is done to cut down on the amount of data that has to be transferred over the network. The JobTracker will then contact TaskTracker Nodes to determine whether the node is available to run a task. The TaskTracker is the service running on the DataNodes that will communicate with the JobTracker for processing tasks it has been assigned.

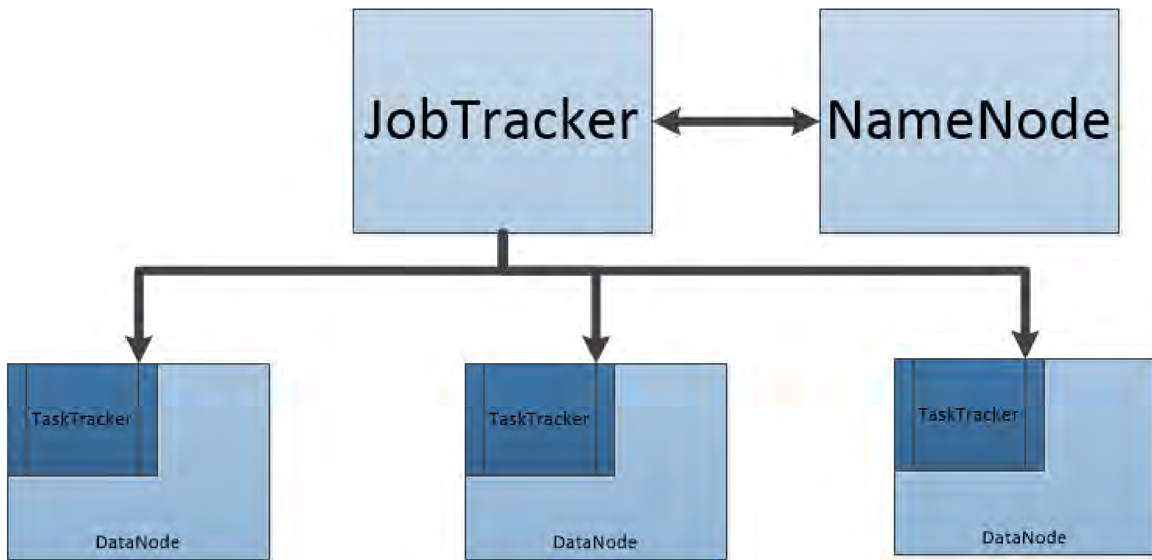


Figure 2.14: HDFS JobTracker Interaction

The DataNode is the true worker in HDFS. The DataNodes are what store and retrieve blocks of data. They inform, the NameNode of the data blocks they have and the JobTracker their status of currents jobs running and/or their availability status. The DataNodes are also the nodes responsible for actually running the MapReduce code on their blocks of data. The DataNodes will communicate with other DataNodes when they need to send or share data. The direct access reduces the amount of traffic needed to be sent if the nodes were required to go though the NameNode to communicate with another DataNode.

CHAPTER 3:
The Experiment Design

The DOD is continuing to create and store all types of data [15]. The technology for how to deal with data is continuously changing. Most current DOD solutions deal with storing data in a relational database and use SQL or Procedural Language/Structured Query Language (PL/SQL) to provide the analytics for this data. SQL and PL/SQL provide a large range of analytics and are very useful for many applications, but when the size of data to analyze becomes large, this approach hits its limitations. Adding a Big Data element to a relational database will provide additional means to store and analyze data.

3.1 Why Add a Big Data Element?

The world is producing petabytes of data daily and the amount of data being stored is increasing. A few examples of this are [13]:

- The New York Stock Exchange generates about one TB of new trade data per day.
- Facebook hosts approximately 10 billion photos, taking up one PB of storage.
- Ancestry.com, the genealogy site, stores around 2.5 PB of data.
- The Internet Archive stores around 2 PB of data, and is growing at a rate of 20 TB per month.
- The Large Hadron Collider near Geneva, Switzerland, will produce about 15 PB of data per year

The large scale of this data makes it difficult to store and process the data in a relational database. A big data element would allow some additional flexibility with storing and analyzing data.

Specifically, the addition of an HDFS cluster would allow quick processing of the entire data set. The true power of Hadoop is that it does process the entire data set [7]. This gives the ability to quickly analyze the entire data set. Hadoop can look at all of the data in the Database and return whatever analysis the programmer desires. Hadoop truly puts the power of all of the data in your corpus at your fingertips. There is little to no concern

that one must perform a large amount of table scans to return analytics; Hadoop by its very nature performs a table scan every time a program is run. That is its true power and it places no limitations on the possible analytics that can be run. Hadoop uses the MapReduce paradigm and forces the programmer to deal with key/value pairs, but these can be chained together to find analytics for all sorts of interesting problems.

3.2 Adding a Big Data Element to GCSS-MC

The current architecture of GCSS-MC does not have an architecture to support big data processing. The architecture that we are purposing in this thesis shows how it can be added as an additional element and integrate with the current architecture as long as it has Internet Protocol (IP) connectivity. The work we did in this thesis shows how a separate HDFS cluster can interact with separate database. We did not have access to the GCSS-MC Database so we used an Oracle Database to simulate the GCSS-MC Database. Figure 3.1 shows the setup the we used for our experimentation.

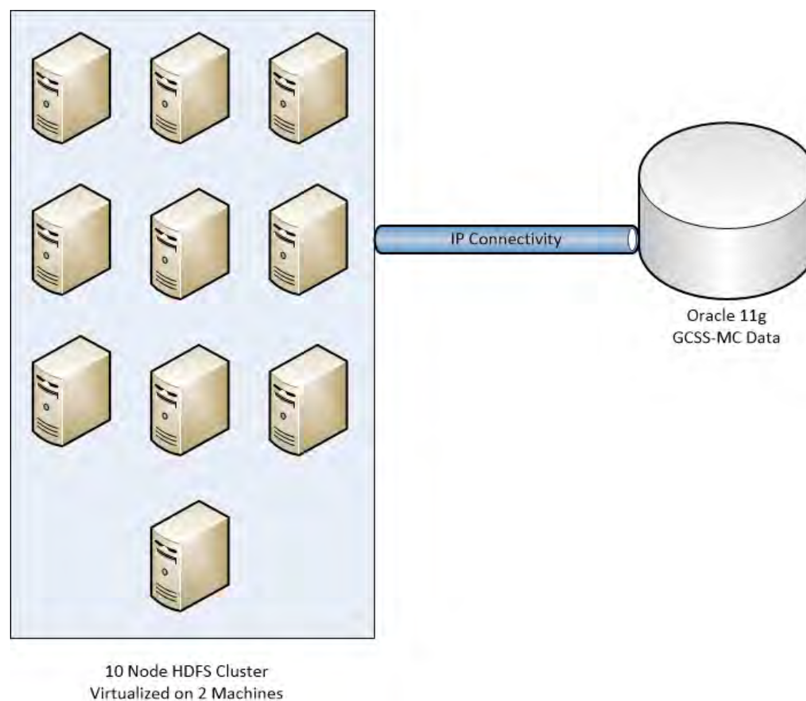


Figure 3.1: Experiment Architecture

This experiment architecture does abstract away some of the difficulty that would be experienced fully integrating a HDFS cluster into the GCSS-MC system. However, we believe that it does fully support the proof of concept that it can be done and done effectively without disrupting the current system. During the setup and testing of this thesis the only stipulation that we found for the cluster to work effectively was that it had to have IP connectivity to the Oracle Database.

The architecture uses is all open source technologies which are readily available to everyone. We choose to use Ubuntu 12.04 LTS for the operating system to run the Hadoop cluster and Oracle 11G as the Relational Database to access and write to because that is what the GCSS-MC system is currently using for their Database. The choices of the operating system and type of database can be changed and adapted to many other situations with a few modifications. That is all of the software that was needed to run the experiment. There are several other Big Data abstractions and tools that can be used, but we felt that keeping the experiment to the core HDFS was important to show this proof of concept and limit the amount of software needed to make the experiment functional.

3.3 Building a Hadoop Cluster

We considered several possibilities discussions on the the best way forward to employ the Hadoop cluster. The initial reaction was to simply run several virtual machines on a large server to represent a cluster. But this thinking goes against the idea of using commodity machines linked together to gain additive power [6]. So we decided to run a simple experiment on the a single Hadoop machine versus a virtualized two node Hadoop cluster. The experiment ran several benchmarks on the two different setups and found that the speed up was hampered by the virtualized environment. Our conclusions came down to no matter how many virtual machines were running ultimately they all had to read and write data to the same hard disk, which subsequently causes a bottleneck. We found some other interesting factors that can be attributed to slower performance in a Hadoop virtual environment as well, the full paper can be found in the Appendix. There has been a lot of research in optimizing Hadoop in the virtual computing environment that has found tunable settings in Hadoop and the virtual hypervisor that can give the same performance as hardware [16].

Despite our findings we decided to run a ten node cluster on two servers. This was largely

decided due to the available recourses and our aim to show a proof of concept (and not focous performance). Once we decided on our way forward we began to build the cluster that would be used for the experiment. The cluster was built on top of two servers running Ubuntu 12.04 LTS. The virtual environment that we choose is Oracle's VirtualBox. The Nodes were split up on the two servers with seven nodes on server one and three on server two. The split was decided based on the available random access memory (RAM) on each server.

Table 3.1: Experiment Server Specifications

Specifications	Server #1	Server #2
OS	Ubuntu 12.04 (64 Bit)	Ubuntu 12.04 (64 Bit)
CPU(s)	4	4
CPU MHZ	1500	1600
HD TB	2	1
RAM GB	32	16
# of HDFS Nodes	7	3

Once the the severs and virtual environment was set up, we built the 10 virtual machine (VM)s on the the servers. The next step was to start and create the Hadoop nodes on the VMs. The Hadoop build was done using Tom White's Hadoop: The Definitive Guide [7] and Michael Noll's Hadoop Tutorials [17]. Each node was built and tested separately prior to adding them to the cluster. This process could have expedited up by cloning the machines, however, we wanted to ensure the integrity of each machine and the cluster.

Table 3.2: Experiment HDFS Node Specifications

Specifications	HDFS Nodes
OS	Ubuntu 12.04 (64 Bit)
CPU(s)	1
HD GB	100
RAM GB	4

The process of installing Hadoop on a VM is not all that different from installing any software package. The Hadoop install is downloaded as a compressed file (.tar.gz). There are a few things that you must do as a prerequisite to the Hadoop install. First, you must ensure that you have an up-to-date version of Java (our testing has found Hadoop works

with Java 6 or Java 7). You must also configure SSH on the VM in order to allow Hadoop to communicate. Once this is done you have to edit several configuration files to set up the environment and then you are off and running. The configuration files are also used to control and optimize the cluster. See the appendix for full installation instructions.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
The Experiment

This chapter will focus on two things: First how the data is taken from and loaded into the Hadoop ecosystem, then how the data is used in two separate analytic programs. The first program, is a home grown analytic tool that uses the GCSS-MC data to find the 100 most frequent NSNs in the whole data set and then build tables based on the top 100 NSNs found. The second program is a simulated SQL program that is taken from a similar project that is asking for SQL analytics on the same data set. The program runs on the data and produces the same results that SQL would produce. The second program is done to show that Hadoop can simulate anything done in an SQL environment. We believe that this makes a strong case that Hadoop can be used to provide analytics on both structured and unstructured data sets.

4.1 The GCSS-MC Data

The GCSS-MC data that we received is a sample of actual production data. This data set was obtained from the USMC and used to assist the thesis experiment. The data was first loaded into an Oracle 11g database. From the database we are able to see fourteen tables of GCSS-MC data. The first step in working with this structured data was to write a program to pull the data out of the relational database and move the data into the Hadoop ecosystem. There were several decisions made as to how to pull the data out and its format. The final choice was made to write a Java program to access the Oracle database using the Java database connect libraries and parse each table into separate files. The program was written to parse each table in the database into JavaScript Object Notation (JSON) format. A sample of the data in JSON format can be found in Figure 4.1. The NSNs are highlighted in the JSON formatted data.

JSON format was chosen because it lends itself nicely to formatting between different databases. Also, a table can quickly be built from the format. This is important because the goal of this project is to pull all of the data from a relational database and run big data analytics and ultimately return back to a simplified relational database table.

```
{
"XXMC_MERIT_RETAIL_INVENTORY": [
1 {"ACTIVITY_ADDRESS_CODE":"MMC100","BACK_ORDER_QUANTITY":"0","CONTROL_ITEM_CODE":"null",
"DAY30_USAGE_RD":"null","DUE_PROVISIONS":"0","DUE_STOCK":"0","EXCH":"0",
"FIXED_LEVEL":"0","FLOAT_REORDER":"25","FREEZE_CODE":"Y","FREEZE_REASON":"GHOST SN",
"FREEZE_DATE":"2013-05-01 23:24:47.0","GABF_DATE":"2013-12-08 11:31:33.0",
"LAST_TRANSACTION_DATE":"2013-10-15 13:24:01.0","MATERIAL_ID_CODE":"B","MOFFSET":"0",
"NON_SYSTEM_ID_CODE":"null","NO_1ST_RECEIPT":"null","OH_PROVISIONS":"0",
"OH_STOCK_SERVICEABLES":"40","OH_UNSERVICEABLES":"0","PHRASE_CODE":"4",
"PRIME_NSN":"4330010463399", "NOMENCLATURE":"FILTER ELEMENT,FLUI", "RECORD_NSN":"4330010463399",
"REORDER_DATE":"2013-12-05 06:07:28.0","REORDER_POINT":"25","REQUISITION_OBJECTIVE":"33",
"ROUTING_ADDRESS_CODE":"MMC300","ROUTING_IDENTIFIER_CODE":"MC1","SEC_CODE":"U",
"SPL_ALLOW":"0","STORE_ACCOUNT_CODE":"1","SUPPLY_SOURCE_CODE":"null","TOTAL_MO_ALLOW":"0",
"UNIT_OF_ISSUE":"EA","UNIT_PRICE":"7.34","PROCESS_STATUS":"Y","RECORD_ID":"73906115",
"CREATED_BY":"1277","CREATION_DATE":"2013-12-08 11:31:33.0","LAST_UPDATED_BY":"1277",
"LAST_UPDATE_DATE":"2013-12-08 11:31:33.0","REQUEST_ID":"27221076","BATCH_ID":"GENEP54431990",
"EXTERNAL_APPLICATION":"merit","REQUIREMENT_CODE":"1FUC","OPERATION_CODE":"61",
"IIP_QUANTITY":"0"},
```


Figure 4.1: Sample GCSS-MC data in JSON Format

There are, however, some compromises that are made when using JSON format. When JSON, is used there is going to be some increase in the size it takes to store that data. In our case the Oracle database is approximately 1.25 gigabyte (GB) and when that data is taken out of the database and parsed into JSON it becomes 5.3 GB. That is a data increase factor of 4.24. This experiment can handle that level of storage increase, but that is not the case as the data becomes larger than 1 TB. We discussed this at length and chose to move forward with JSON with the understanding that this would have to change to increase the scale of the data. The main reason for us to maintain JSON is because keeping the meta-data in the data provides more flexibility in transforming and exporting the data dynamically.

Another option would have been to use a different format for the data that would reduce the size requirements. For example, the data could have been pulled from the database and parsed into Comma Separated Value (CSV) format. The CSV format would reduce the data blow up factor significantly. In the data set for this thesis, the increase factor for CSV would be 1.176, which is 4.5 times less than JSON format. Additionally, the data could be reduced when the database has a null value. For this dataset we did not reduce the null values out of the data. Again, the main reason to keep the null value was to provide the most flexibility to the analytics in the future.

There has to be a great deal of time spent on how to deal with the data when the possibility for the data to exceed TBs exists. The larger the data set, the more effort must be spent in

minimizing the size of the data. Especially when the Hadoop ecosystem will increase the storage requirement of any dataset by three. This is due to the data replication factor set for Hadoop. The data replication factor is tunable but if reduced below three, the cluster will be more susceptible to faults and may decrease speed of the programs due to the larger network overhead of having to send the data to a node to run the computation. The Hadoop NameNode will try to schedule the data computation on an idle node where the data resides. Therefore, if the replication factor is decreased then the chances of finding an idle node that has the data decreases. Thus the NameNode will have to schedule the computation to another node and send that node the data resulting in the overhead of additional scheduling message as well as the actual sending of the data on the network. Hadoop can handle large amounts of data and the main limit on the size of the data Hadoop can handle is the physical limitations of the machines on which it is deployed. As an illustration to the possibility of a Hadoop cluster; in 2010, Facebook had a Hadoop cluster that was 2,000 nodes and had a storage capacity of twenty-one PB [18].

Once all of the decisions were made on how to parse the GCSS-MC database, the next step was importing the data into the actual Hadoop ecosystem. We chose to simply use the Hadoop file manipulations commands to ingest the JSON feed into the the Hadoop ecosystem. The ingest is ultimately accomplished with a bash script executed on the NameNode. The bash script executes the Java program to pull the data from the Oracle database and then executes the Hadoop file system command to import the data. We experimented with Squoop to do this and were pleased with the results. Squoop is an Apache Hadoop tool that supports importing and exporting of database data into Hadoop. However, because Squoop did not support the JSON format we needed, we wrote a Java program that executes Java Database Connectivity (JDBC) calls to the database and then generates the JSON format. The program is designed to make the JDBC and call to the database and create the JSON format for each row on a separate line in a text file. Each row of data in the database representing one line in the text file is important because of how Hadoop reads the file in a MapReduce algorithm. For instance, if the program spread a row of data beyond one line in the value, it would be near impossible to write a MapReduce program to recreate the database row. This is due to Hadoop MapReduce handling each line of input data separately, which is extremely important to ensure the program can be split up and executed in parallel. Of course, we could write another program to handle that and recreate the row,

but that would incur a large overhead cost. Because we wrote the program that pulls that data, it gives us the power to control how the data is handled. We do not mean to say using additional tools like Squoop or another third party tool is a bad thing, just that in building our program we wanted to ensure we controlled the data at each stage and writing our own program to handle that was the best solution in our experiment. Additionally, we decided to try and keep the smallest footprint of software that was needed to run this experiment. We thought that if we can create all of the programs we needed, we would better understand the data flow and thus better understand how to use the tools that abstract the lower level coding.

4.2 Top-100-NSN Program

The first program we designed and developed is the NSN program. This program scans all 13 files pulled from the database, and finds the top 100 most frequently occurring NSNs. Then it pulls the data associated with those NSNs and creates a SQL statement to write the data back to the database. The idea of this program was to scan a large dataset and then create smaller tables that contain only specifically defined needed data. During the creation of this program we made the choices as to what data to return. Although, we made educated guesses on what a Supply Officer would deem important, that is not the true proof of importance of what we were able to show. The fact that the data is returned is what is important. The choice of what subset of data is returned does not matter; that can be changed in the program and then whatever data a customer wants can be delivered.

In order to get the desired result the program had to be written to execute several MapReduce jobs chained together. In this program we had to write five separate MapReduce algorithms in order to achieve our desired result. The first MapReduce algorithm simply counts the occurrences of NSNs. The second algorithm sorts the NSNs in descending order of greatest frequency. The next algorithm finds all of the data associated with the NSNs. Then the final two algorithms create the JSON and the SQL statements and write the data to the Oracle database. Each one of the algorithms was individually created and tested. Once all of the algorithms were completed, we packaged them up into a single MapReduce job to run.

The flow of the entire program is illustrated in Figure 4.2. The figure gives a graphic

representation of the data flow through the whole program by illustrating the data at each algorithm. The inputs are on the left side of the algorithm boxes and the output at the right side. The inputs to the top of the box are used by the algorithm in the configure method of the map phase.

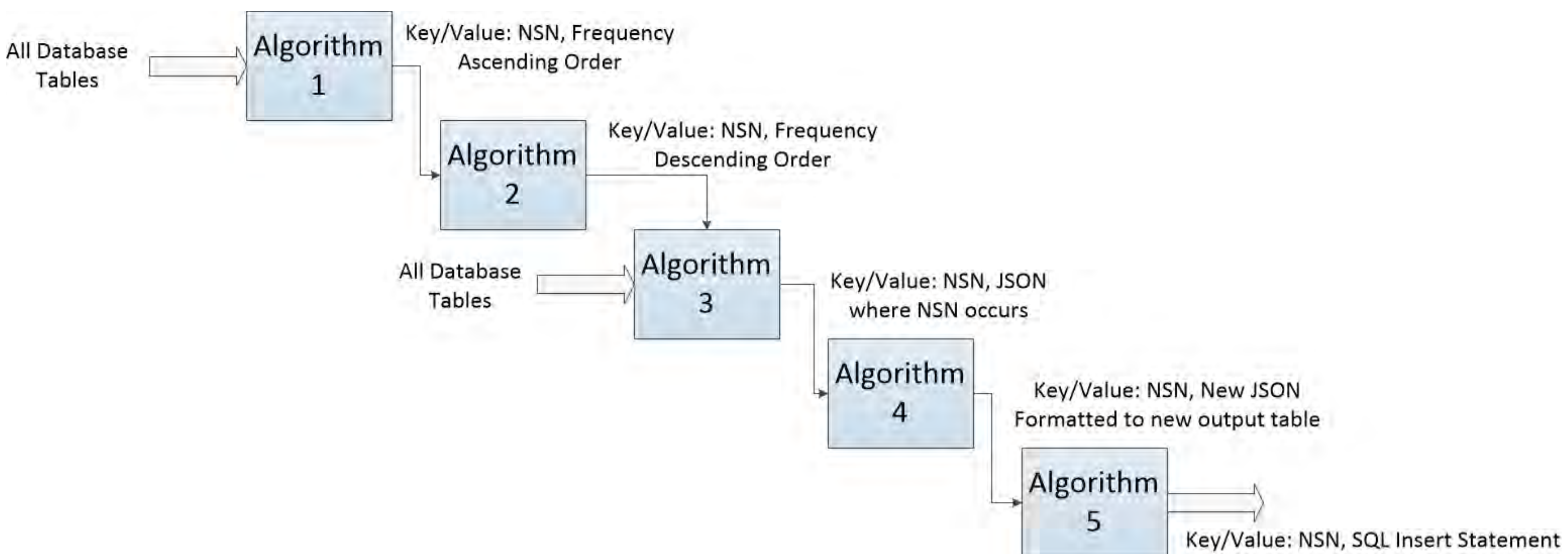


Figure 4.2: Data Flow through the NSN Program

4.2.1 Top-100-NSN Program: First Algorithm

The first MapReduce algorithm in the NSN program finds and counts all of the NSNs in the entire corpus. This algorithm is similar to a word count algorithm.

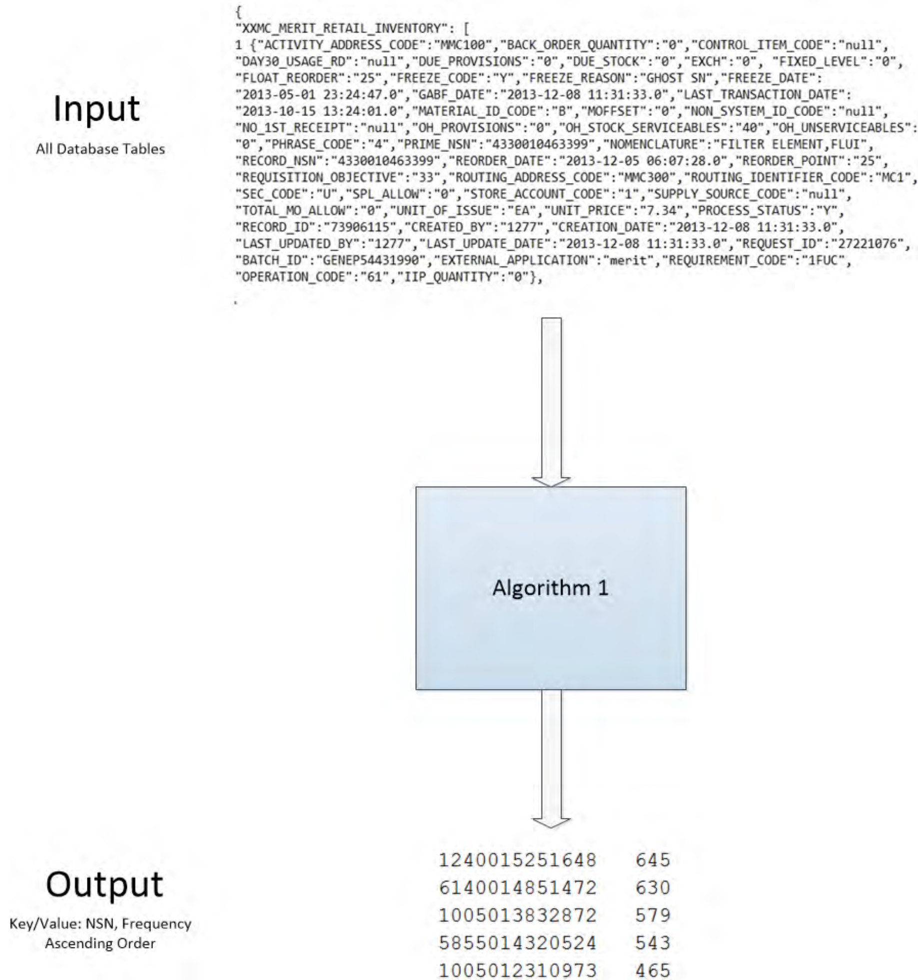


Figure 4.3: NSN: First Algorithm

The major difference is the "word" that is counted is limited to an NSN. The map phase scans all thirteen tables and outputs only a valid NSN. The reduce phase will then sum up all of the occurrences of the NSNs and produce the output. This step is the first in the analysis process and is needed to find and output the most frequent NSNs in the data corpus. The data input to the algorithm is similar to the data in Figure 4.1 and the highlighted values are the NSNs we are searching for in this algorithm. The output sample is shown in Figure

4.4.

```
1240015251648    645
6140014851472    630
1005013832872    579
5855014320524    543
1005012310973    465
```

Figure 4.4: NSN Output from the First Algorithm

4.2.2 Top-100-NSN Program: Second Algorithm

The next step is to sort the NSNs by frequency. Hadoop will always provide some order on the output from the reduce phase. The order is given based on the value of the key of the key/value pair. Hadoop, by default, will order the output of the reduce phase in a alphabetic and numeric ascending order.

In this algorithm we want to have the reduce phase produce an output that will be ordered by the frequency of NSN occurrences in descending order. This has to be done in a separate algorithm than the first. If we attempt to order in the first algorithm before it completes, then the sum of the NSN occurrences will not be accurately produced. Now we have the output of the first algorithm as the input into the second algorithm. There are two major obstacles to overcome to produce the desired output. First we have to override the output key comparator class to produce a key output in descending order. This process occurs in the combining and shuffling steps of Hadoop. The code created to perform this descending order sort is shown in Figure 4.6. The method overrides the method Hadoop calls to compare. This method is a recursive function that transposes the order by multiplying everything by a negative one.

The second obstacle is that we need to transpose the key/value pairs such that the frequency value of NSNs is now the key and the value becomes the NSN. This is done to produce an

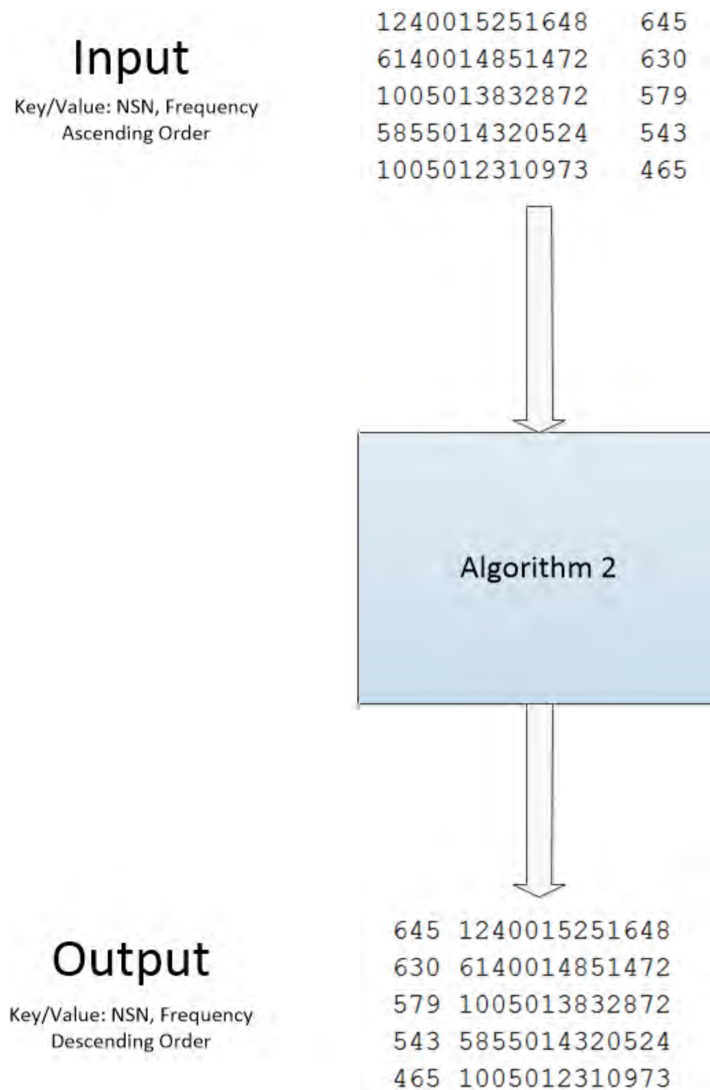


Figure 4.5: NSN: Second Algorithm

output that will be ordered in descending order based on the frequency of occurrence of a particular NSN. The transposing of the key/value pair is accomplished in the map phase. Then the shuffling and combing phase will produce the input to the reduce phase with the NSN frequency as the key and the NSN itself the value. A sample of the output of the algorithm can be seen in Figure 4.7.

```
1  static class ReverseComparator extends WritableComparator {
2          private static final Text.Comparator TEXT_COMPARATOR = new Text.Comparator();
3
4          public ReverseComparator() {
5              super(Text.class);
6          }
7
8          @Override
9          public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
10                 return (-1)* TEXT_COMPARATOR
11                         .compare(b1, s1, l1, b2, s2, l2);
12         }
13
14     }
```

Figure 4.6: Descending Order Sort Class

```
645 1240015251648
630 6140014851472
579 1005013832872
543 5855014320524
465 1005012310973
```

Figure 4.7: NSN Output from the Second Algorithm

4.2.3 Top-100-NSN Program: Third Algorithm

The output from the second algorithm enables us to search and find all of the data associated with the 100 most frequently occurring NSNs. In order to facilitate using the output from the second algorithm we used the distributed cache functionality that is part of Hadoop. The distributed cache allows the programmer to access other files on the HDFS. Then we had to decide what type of data structure to use to build with the output from algorithm two. After some initial testing we found that using a hash map provided the faster comparisons than a map or a pattern. Hadoop also provides the configure method to use as means to set up data structures that will be needed in the map phase. The configure method allow the

programmer to build data structures that can be used in the map phase of the algorithm. This is critical because each map instance will have to build the data structure exactly the same in order to achieve predictable,repeatable results. In this particular configure method we create a hash map with the first 100 NSNs from the output of algorithm two.

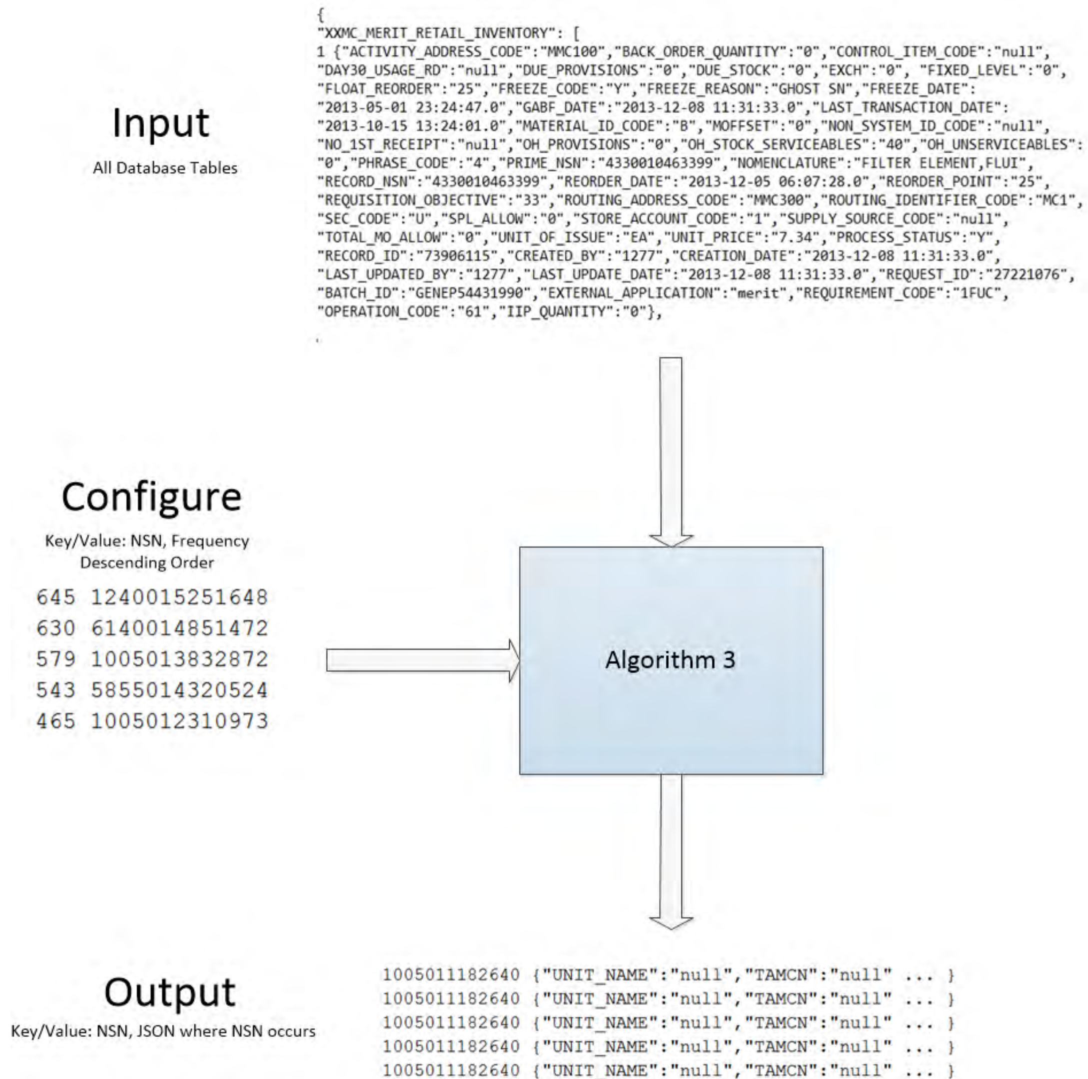


Figure 4.8: NSN: Third Algorithm

The map phase will then scan all of the data in the corpus and produce an output from the map phase if and only if the NSN is found in the line of data. The line is output in JSON format with the NSN and the source file appended to the front of the JSON. The

reduce phase of this algorithm is the identity reduce. It simply outputs the key/value pairs it receives from the map phase. A sample of the output can be found in Figure 4.9.

```
1005011182640 {"UNIT_NAME":"null","TAMCN":"null" ... }
1005011182640 {"UNIT_NAME":"null","TAMCN":"null" ... }
1005011182640 {"UNIT_NAME":"null","TAMCN":"null" ... }
1005011182640 {"UNIT_NAME":"null","TAMCN":"null" ... }
1005011182640 {"UNIT_NAME":"null","TAMCN":"null" ... }
```

Figure 4.9: NSN Output from the Third Algorithm

4.2.4 Top-100-NSN Program: Fourth Algorithm

The next step is to use the output from the third algorithm and parse the format down to the JSON that we wish to write back to the database.

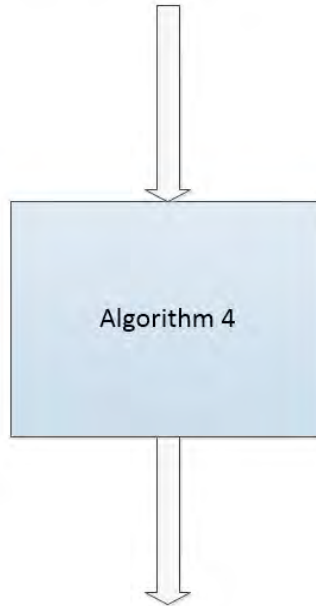
This algorithm and the next one could be reduced to one that generates the JSON and creates the SQL statement to write to the database using a JDBC call. Ultimately, we decided to keep them separate so that we could use the JSON format in the future to regenerate the tables, if necessary. Furthermore, to maintain the flexibility to adapt to other tools, like a database Hadoop connector, we felt it prudent to keep the JSON formating code. This algorithm takes the output from the third algorithm and parses it down into a JSON format to write back to the database. A sample of the output of the algorithm can be seen in Figure 4.11.

At this point we made some decisions about what data to write back. Table 4.1 illustrates the data that we write back to the database. At this point the data choice can be modified and tailored to exactly what a customer/stakeholder would desire. The choices were made just show a proof of concept that the data can be found and written back to a database to show a smaller amount of data that is desired. However, any of the data that exists in the original database can be retrieved with only minor modifications to the code base. The benefit of this type of analytic is truly realized as the data size increases, the customer can keep all of their data and produce tables of just the pertinent data required for a particular need.

Input

Key/Value: NSN, JSON where NSN occurs

1005011182640 {"UNIT_NAME":"null","TAMCN":"null" ... }
1005011182640 {"UNIT_NAME":"null","TAMCN":"null" ... }
1005011182640 {"UNIT_NAME":"null","TAMCN":"null" ... }
1005011182640 {"UNIT_NAME":"null","TAMCN":"null" ... }
1005011182640 {"UNIT_NAME":"null","TAMCN":"null" ... }



Output

Key/Value: NSN, New JSON
Formatted to new output table

{"NSN":"1005011182640" Table Name: XXMC_R001_ALLOWANCES_TBL.TXT ,"UNIT_NAME":"NULL" ...}
{"NSN":"1005011182640" Table Name: XXMC_R001_ALLOWANCES_TBL.TXT ,"UNIT_NAME":"NULL" ...}
{"NSN":"1005011182640" Table Name: XXMC_R001_DUEIN_TBL.TXT ,"ORDER_NUMBER":"1041015" ...}
{"NSN":"1005011182640" Table Name: XXMC_R001_DUEIN_TBL.TXT ,"ORDER_NUMBER":"1596718" ...}
{"NSN":"1005011182640" Table Name: XXMC_R001_DUEIN_TBL.TXT ,"ORDER_NUMBER":"616556" ...}

Figure 4.10: NSN: Fourth Algorithm

{"NSN":"1005011182640" Table Name: XXMC_R001_ALLOWANCES_TBL.TXT ,"UNIT_NAME":"NULL" ...}
{"NSN":"1005011182640" Table Name: XXMC_R001_ALLOWANCES_TBL.TXT ,"UNIT_NAME":"NULL" ...}
{"NSN":"1005011182640" Table Name: XXMC_R001_DUEIN_TBL.TXT ,"ORDER_NUMBER":"1041015" ...}
{"NSN":"1005011182640" Table Name: XXMC_R001_DUEIN_TBL.TXT ,"ORDER_NUMBER":"1596718" ...}
{"NSN":"1005011182640" Table Name: XXMC_R001_DUEIN_TBL.TXT ,"ORDER_NUMBER":"616556" ...}

Figure 4.11: NSN Output from the Fourth Algorithm

Table 4.1: Tables Written Back to Database

Table Name	Source Table Name	Column #1	Column #2	Column #3	Column #4	Column #5
HDFS_MT_RETAIL_INV_NSN	XXMC_MERIT_RETAIL_INVENTORY	NSN	RECORD_ID	ACTIVITY_ADDRESS_CODE	NOMENCLATURE	OH_STOCK_SERVICEABLES
HDFS_R001_ALLOWANCES_NSN	XXMC_R001_ALLOWANCES_TBL	NSN	RECORD_ID	REQUIREMENT_CODE	UNIT_NAME	ORGANIZATIONAL_CODE
HDFS_R001_DUEIN_NSN	XXMC_R001_DUEIN_TBL	NSN	RECORD_ID	DOCUMENT_NUMBER	ORDER_QTY	ORDER_NUMBER
HDFS_R001_HIST_DUEIN_NSN	XXMC_R001_HIST_DUEIN_TBL	NSN	RECORD_ID	DOCUMENT_NUMBER	ORDER_QTY	ORDER_NUMBER
HDFS_R001_DUEIN_STAT_NSN	XXMC_R001_DUEIN_STAT_TBL	NSN	RECORD_ID	DOCUMENT_NUMBER	DOC_IDENT_NUMBER	STATUS_CODE
HDFS_R001_HIST DUEIN_ST_NSN	XXMC_R001_HIST_DUEIN_ST_TBL	NSN	RECORD_ID	DOCUMENT_NUMBER	DOC_IDENT_NUMBER	STATUS_CODE
HDFS_R001_INVENTORY_NSN	XXMC_R001_INVENTORY_TBL	NSN	RECORD_ID	STATUS_CODE	SERIAL_NUMBER	TAMCN
HDFS_R001_ITEMMASTER_NSN	XXMC_R001_ITEMMASTER_TBL	NSN	RECORD_ID	ACQUISITION_ADVICE_CODE	NOMENCLATURE	MANAGEMENT_ECHELON_CODE
HDFS_R001_REPAIRPARTS_NSN	XXMC_R001_REPAIRPARTS_TBL	NSN	RECORD_ID	ORG_CODE	DOCUMENT_NUMBER	DEMAND_CODE
HDFS_R001_SRHEADERS_NSN	XXMC_R001_SRHEADERS_TBL	NSN	RECORD_ID	SERVICE_REQUEST_TYPE	SR_NUMBER	PROBLEM_SUMMARY
HDFS_R001_SRTASKS_NSN	XXMC_R001_SRTASKS_TBL	NSN	RECORD_ID	TASK_NUMBER	TASK_NAME	TASK_STATUS

4.2.5 Top-100-NSN-Program: Fifth Algorithm

The final algorithm for this program performs the MapReduce function that makes the JDBC call to write the data back to the database. This algorithm takes the formated JSON output from the previous algorithm and creates an SQL statement to write the data to the database.

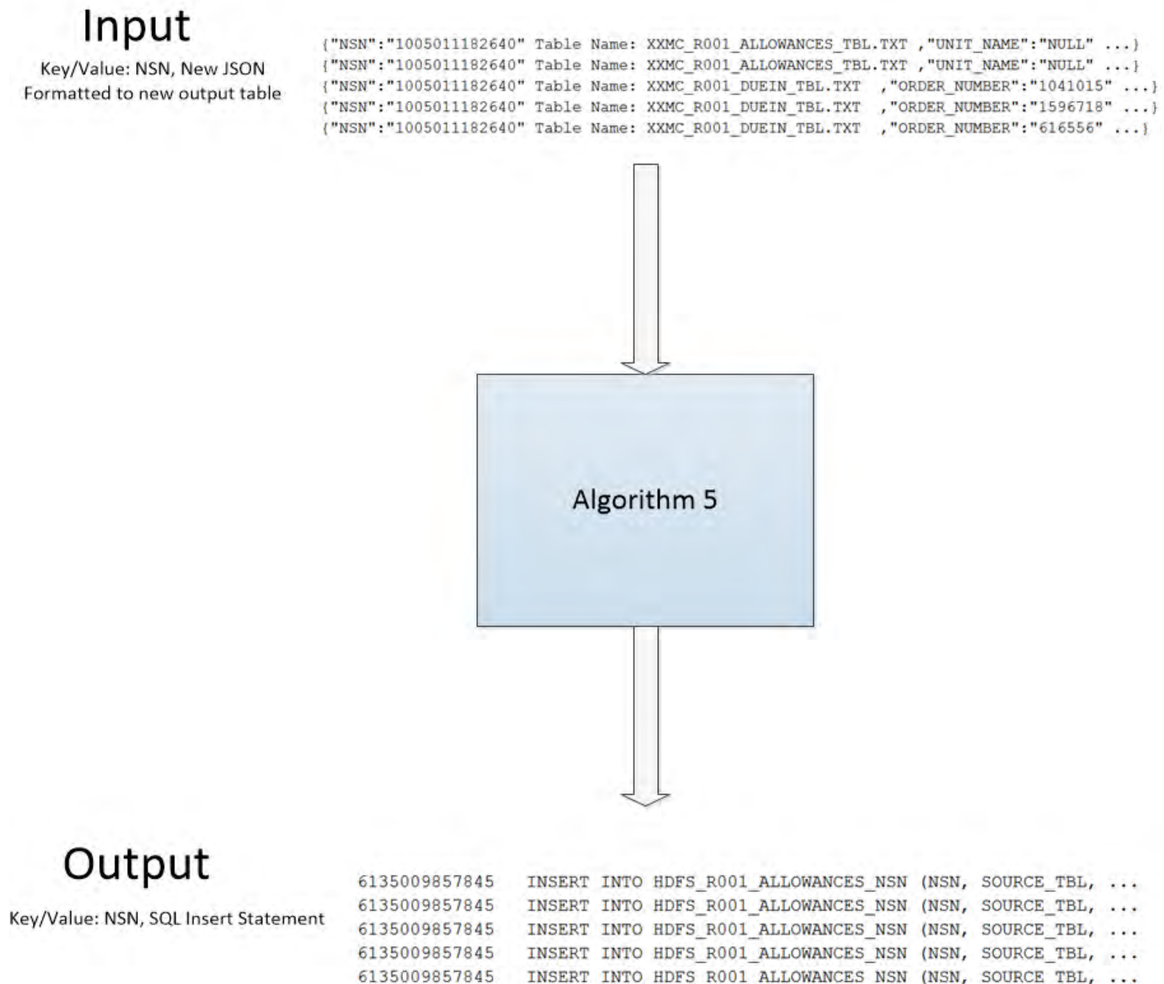


Figure 4.12: NSN: Fifth Algorithm

The JDBC connection is made in the configure method of the map phase. The map phase then creates the statement and makes the JDBC call to execute the statement. The output of the map class is the NSN and SQL statement. The reduce phase in the identity reducer and

simply outputs what it receives. The reduce phase could be eliminated and the output taken directly from the Map phase, which would speed up the algorithm execution. We decided to keep the reduce phase to sort the map output. A sample output of the algorithm can be seen in Figure 4.13.

```
6135009857845    INSERT INTO HDFS_R001_ALLOWANCES_NSN (NSN, SOURCE_TBL, ...
6135009857845    INSERT INTO HDFS_R001_ALLOWANCES_NSN (NSN, SOURCE_TBL, ...
6135009857845    INSERT INTO HDFS_R001_ALLOWANCES_NSN (NSN, SOURCE_TBL, ...
6135009857845    INSERT INTO HDFS_R001_ALLOWANCES_NSN (NSN, SOURCE_TBL, ...
6135009857845    INSERT INTO HDFS_R001_ALLOWANCES_NSN (NSN, SOURCE_TBL, ...
```

Figure 4.13: NSN Output from the Fifth Algorithm

4.2.6 Top-100-NSN Program Conclusion

The result of the entire NSN program is the creation of eleven new tables in the database with an aggregated 916,190 rows. Table 4.2 shows the first ten rows of data form the `HDFS_R001_INVENTORY_NSN` table create by the NSN program. The power of this type of analytic is that it gives the customer the power to analyze all data and filter their data down to only the data they are concerned with at the time. This creates the flexibility to maintain and analyze their entire data set and also reduce it down to a manageable subsets of needed data. This program does just that; it scans all of the data and then returns to the database only the data of the most frequent NSNs with the reduced subset of the original data. Although, this program writes a subset of the data back to the database, it just as easily could reduce the data further into fewer tables by joining the data from any/all of the tables. The possibility of the analytics Hadoop can perform is limitless.

Table 4.2: Sample of HDFS_R001_INVENTORY_NSN Table

NSN	Source Table Name	RECORD_ID	STATUS_CODE	SERIAL_NUM	TAMCN
1005013832872	XXMC_R001_INVENTORY_TBL	441912925	LATEST	10328808	E14422M
1005013832872	XXMC_R001_INVENTORY_TBL	441912926	LATEST	10328831	E14422M
1005013832872	XXMC_R001_INVENTORY_TBL	441912927	LATEST	10328995	E14422M
1005013832872	XXMC_R001_INVENTORY_TBL	441912928	CREATED	10329091	E14422M
1005013832872	XXMC_R001_INVENTORY_TBL	441912929	LATEST	10329141	E14422M
1005013832872	XXMC_R001_INVENTORY_TBL	441912930	LATEST	10329204	E14422M
1005013832872	XXMC_R001_INVENTORY_TBL	441912931	LATEST	10329312	E14422M
1005013832872	XXMC_R001_INVENTORY_TBL	441912932	LATEST	10329315	E14422M
1005013832872	XXMC_R001_INVENTORY_TBL	441912933	LATEST	10329316	E14422M
1005013832872	XXMC_R001_INVENTORY_TBL	441912934	LATEST	10329351	E14422M

4.3 SQL Simulation

The SQL simulation program was an analytic taken from another ongoing Naval Postgraduate project that is examining the same GCSS-MC data. We felt it prudent to create an analytic on something the USMC is looking for currently. The other project is focusing on using SQL to generate analytics and then display the results in a web tier architecture. Although we do not take the results to a display in the web tier, we do produce the same results and write the results back to the database. The overarching idea is that the HDFS resources are used to process the data and then something like a web tier business analytic suite can display the table in a graph. The total program, we will call it Alpha, consists of four MapReduce algorithms. The purpose of the analytic is to produce a readiness report aggregated over time, equipment, and/or unit. Figure 4.14 illustrates the data relationship and the logic used in the Alpha program.

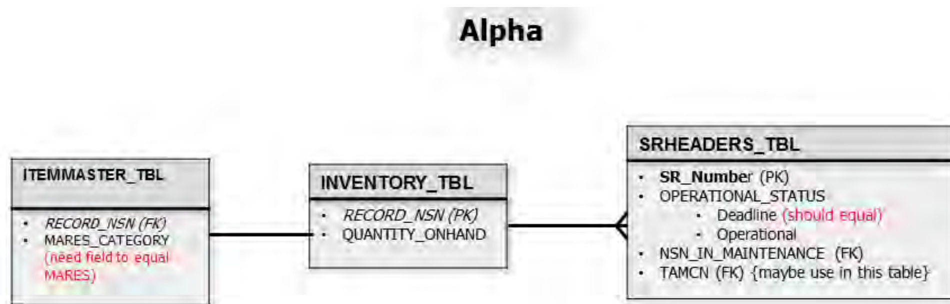


Figure 4.14: Alpha Program Logic

The flow of the entire program is illustrated in Figure 4.15. The figure gives a graphic representation of the data flow through the whole program by illustrating the data at each algorithm. The inputs are on the left side of the algorithm boxes and the output at the right side. The inputs to the top of the box are used by the algorithm in the configure method of the map phase.

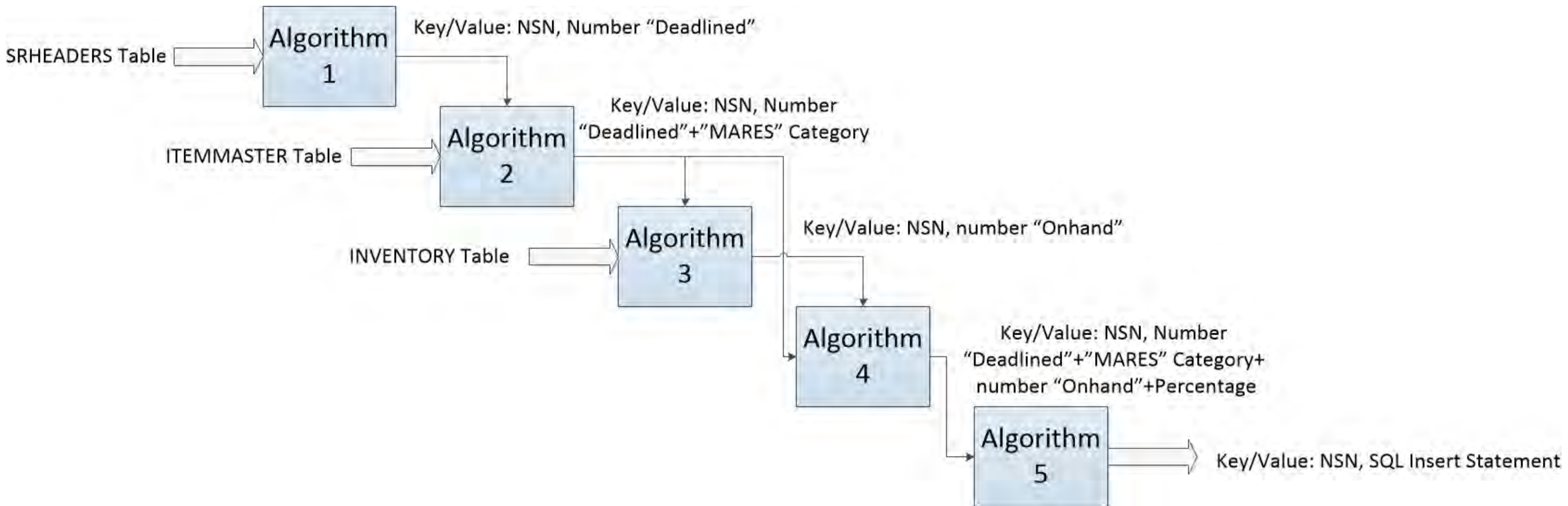


Figure 4.15: Data Flow Through the Alpha Program

The SQL analytic consists of data calculations and three table joins. Accomplishing this in Hadoop requires the five MapReduce steps. The first step is counting all NSNs with an Operational Status that are "Deadlined" value in the XXMC_R001_SRHEADERS_TBL. The next step scans the XXMC_R001_ITEMMASTER_TBL and gets all of the NSNs that are "MARES" reportable. The third step scans the XXMC_R001_INVENTORY_TBL and calculates all the number of "Onhand" NSNs. The fourth step joins the outputs from step two and three and performs the calculation for percentage of "Deadlined" versus "Onhand". The final step writes the data back to the database table.

4.3.1 SQL Simulation: First Algorithm

The first MapReduce algorithm performed in the alpha program will scan the XXMC_R001_SRHEADERS_TBL. The map phase will parse out the data and find all of the rows that have an Operational Status of "Deadlined". Once the "Deadlined" value is seen in the row the NSN associated with that row is recorded and then the map phase will output the key/value pair of the NSN and the value one. The SRHEADERS table will have and entry for each item that has been "Deadlined". Therefore, there may be several items "Deadlined" associated with a particular NSN and we need to have the total number "Deadlined" for each NSN.

The reduce phase will take care of summing up all of the "Deadlined" values for each NSN. The output from the first algorithm is the NSN and the total number of items "Deadlined" for that particular NSN. A sample of the output of the algorithm can be seen in Figure 4.17.

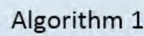
4.3.2 SQL Simulation: Second Algorithm

The second MapReduce algorithm will take as input the XXMC_R001_ITEMMASTER_TBL. The alogrithm will also access the output from the first algorithm as distributed cache, Figure 4.15, and will be used in the configure method to build a hash map with the key/value pair as the NSN/number "Deadlined". The map phase will then scan the ITEMMASTER table and check all NSNs to determine the "MARES" status. If the NSN is "MARES" reportable, the algorithm will check to see if the NSN is a key in the hash map. If the NSN is a key in the hash map then the number "Deadlined" will be obtained from the hash map and the map phase will set the output key/value pair to NSN/"MARES"_"Number

Input

SRHEADERS Table

{
1 {"ACTIVITY_ADDRESS_CODE":"MMC100","BACK_ORDER_QUANTITY":"0","CONTROL_ITEM_CODE":"null",
"DAY30_USAGE_RD":"null","DUE_PROVISIONS":"0","DUE_STOCK":"0","EXCH":"0",
"FIXED_LEVEL":"0","FLOAT_REORDER":"25","FREEZE_CODE":"Y","FREEZE_REASON":"GHOST SN",
"FREEZE_DATE":"2013-05-01 23:24:47.0","GABF_DATE":"2013-12-08 11:31:33.0",
"LAST_TRANSACTION_DATE":"2013-10-15 13:24:01.0","MATERIAL_ID_CODE":"B","MOFFSET":"0",
"NON_SYSTEM_ID_CODE":"null","NO_1ST_RECEIPT":"null","OH_PROVISIONS":"0",
"OH_STOCK_SERVICEABLES":"40","OH_UNSERVICEABLES":"0","PHRASE_CODE":"4",
"PRIME_NSN":"4330010463399", "NOMENCLATURE":"FILTER ELEMENT,FLUI", "RECORD_NSN":"4330010463399",
"REORDER_DATE":"2013-12-05 06:07:28.0","REORDER_POINT":"25","REQUISITION_OBJECTIVE":"33",
"ROUTING_ADDRESS_CODE":"MMC300","ROUTING_IDENTIFIER_CODE":"MC1","SEC_CODE":"U",
"SPL_ALLOW":"0","STORE_ACCOUNT_CODE":"1","SUPPLY_SOURCE_CODE":"null","TOTAL_MO_ALLOW":"0",
"UNIT_OF_ISSUE":"EA","UNIT_PRICE":"7.34","PROCESS_STATUS":"Y","RECORD_ID":"73906115",
"CREATED_BY":"1277","CREATION_DATE":"2013-12-08 11:31:33.0","LAST_UPDATED_BY":"1277",
"LAST_UPDATE_DATE":"2013-12-08 11:31:33.0","REQUEST_ID":"27221076","BATCH_ID":"GENEP54431990",
"EXTERNAL_APPLICATION":"merit","REQUIREMENT_CODE":"1FUC","OPERATION_CODE":"61",
"IIP_QUANTITY":"0"},



Output

Key/Value: NSN, Number “Deadlined”

```
1005007265636   187
1005009573893   9
1005010258095   21
1005010351674   1
1005011055191   1
```

Figure 4.16: Alpha: First Algorithm

Deadlined".

The reduce phase in this case is the identity reducer. A sample of the output of the algorithm can be seen in Figure 4.19.

```
1005007265636   187
1005009573893   9
1005010258095   21
1005010351674   1
1005011055191   1
```

Figure 4.17: Alpha Output from the First Algorithm

4.3.3 SQL Simulation: Third Algorithm

The next MapReduce algorithm takes the `XXMC_R001_INVENTORY_TBL` as input. The algorithm will use the output from the second algorithm to create a hash map in the configure method. The hash map will store the NSN as the key and the value will be a string that contains the "Mares" category and number "Deadlined". The map phase is going to scan the INVENTORY table and check if the NSN is in the hash map and if it is then get the value for the quantity "Onhand". The map output is the key/value pair NSN/number "Onhand".

The reduce phase will calculate the total number "Onhand" for each NSN. A sample output of the algorithm can be seen in Figure 4.21.

4.3.4 SQL Simulation: Fourth Algorithm

The fourth MapReduce algorithm takes the output from the second algorithm as input and the output from the third algorithm as distributed cash. The configure method is used to build a hash map from the third algorithm output. The map phase then performs a map side join on the output from the previous two algorithms. The input data is parsed and then values from the hash map are appended. The algorithm is also responsible for performing the percentage of "Deadlined" vs. number "Onhand". The map output is the key/value pair NSN/number "Deadlined"_"MARES" Category_number "Onhand"_Percentage.

The reduce phase is the identity reducer. A sample of the output of the algorithm can be

Input

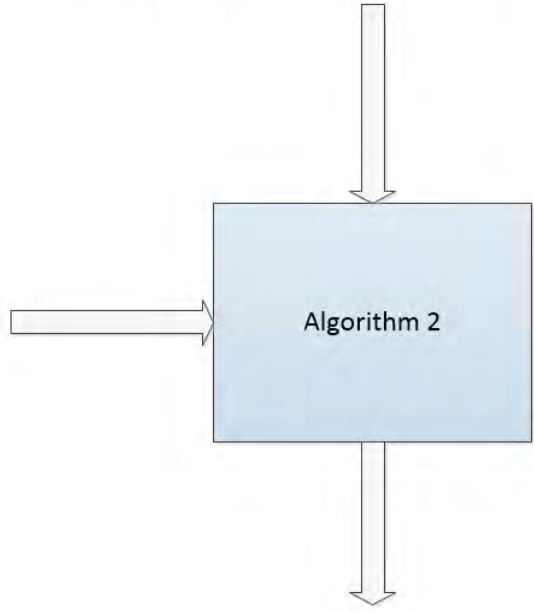
ITEMMASTER Table

{
1 {"ACTIVITY_ADDRESS_CODE":"MMC100","BACK_ORDER_QUANTITY":"0","CONTROL_ITEM_CODE":"null",
"DAY30_USAGE_RD":"null","DUE_PROVISIONS":"0","DUE_STOCK":"0","EXCH":"0",
"FIXED_LEVEL":"0","FLOAT_REORDER":"25","FREEZE_CODE":"Y","FREEZE_REASON":"GHOST SN",
"FREEZE_DATE":"2013-05-01 23:24:47.0","GABF_DATE":"2013-12-08 11:31:33.0",
"LAST_TRANSACTION_DATE":"2013-10-15 13:24:01.0","MATERIAL_ID_CODE":"B","MOFFSET":"0",
"NON_SYSTEM_ID_CODE":"null","NO_1ST_RECEIPT":"null","OH_PROVISIONS":"0",
"OH_STOCK_SERVICEABLES":"40","OH_UNSERVICEABLES":"0","PHRASE_CODE":"4",
"PRIME_NSN":"4330010463399", "NOMENCLATURE":"FILTER ELEMENT,FLUI", "RECORD_NSN":"4330010463399",
"REORDER_DATE":"2013-12-05 06:07:28.0","REORDER_POINT":"25","REQUISITION_OBJECTIVE":"33",
"ROUTING_ADDRESS_CODE":"MMC300","ROUTING_IDENTIFIER_CODE":"MC1","SEC_CODE":"U",
"SPL_ALLOW":"0","STORE_ACCOUNT_CODE":"1","SUPPLY_SOURCE_CODE":"null","TOTAL_MO_ALLOW":"0",
"UNIT_OF_ISSUE":"EA","UNIT_PRICE":"7.34","PROCESS_STATUS":"Y","RECORD_ID":"73906115",
"CREATED_BY":"1277","CREATION_DATE":"2013-12-08 11:31:33.0","LAST_UPDATED_BY":"1277",
"LAST_UPDATE_DATE":"2013-12-08 11:31:33.0","REQUEST_ID":"27221076","BATCH_ID":"GENEP54431990",
"EXTERNAL_APPLICATION":"merit","REQUIREMENT_CODE":"1FUC","OPERATION_CODE":"61",
"IIP_QUANTITY":"0"},

Configure

Key/Value: NSN, Number “Deadlined”

```
1005007265636    187
1005009573893    9
1005010258095    21
1005010351674    1
1005011055191    1
```



Output

Key/Value: NSN, Number “Deadlined”_”MARES” Category

```
1005007265636    187_MARES
1005009573893    9_MARES
1005010351674    1_MARES
1005013592714    0_MARES
1005014123129    172_MARES
```

Figure 4.18: Alpha: Second Algorithm

seen in Figure 4.23.

4.3.5 SQL Simulation: Fifth Algorithm

The final MapReduce algorithm will take the output from the fourth algorithm as input. The configure method will be used to set up the JDBC connection to the database. The map phase parses the input and creates and executes a SQL insert statement for each line of input.

```
1005007265636    187_MARES
1005009573893    9_MARES
1005010351674    1_MARES
1005013592714    0_MARES
1005014123129    172_MARES
```

Figure 4.19: Alpha Output from the Second Algorithm

The final output of the algorithm is the key/value pair of NSN/SQL statement.A sample output of the algorithm can be seen in Figure 4.25.

```
1005007265636    187
1005009573893    9
1005010258095    21
1005010351674    1
1005011055191    1
```

Figure 4.25: Alpha Output from the Fifth Algorithm

4.3.6 Alpha Program Conclusion

The result of the entire program is a join across three tables, calculations, and a database table created that contains all of the data. A sample of the resultant Alpha table is in Table 4.3. The purpose of this analytic was to show that Hadoop can perform the same type of analysis of structured data that SQL can. This will become increasingly important as the data set approaches values over 1TB. The methodology can be adapted to perform all types of SQL statements with no limitation. One thing the Alpha program does not show is the

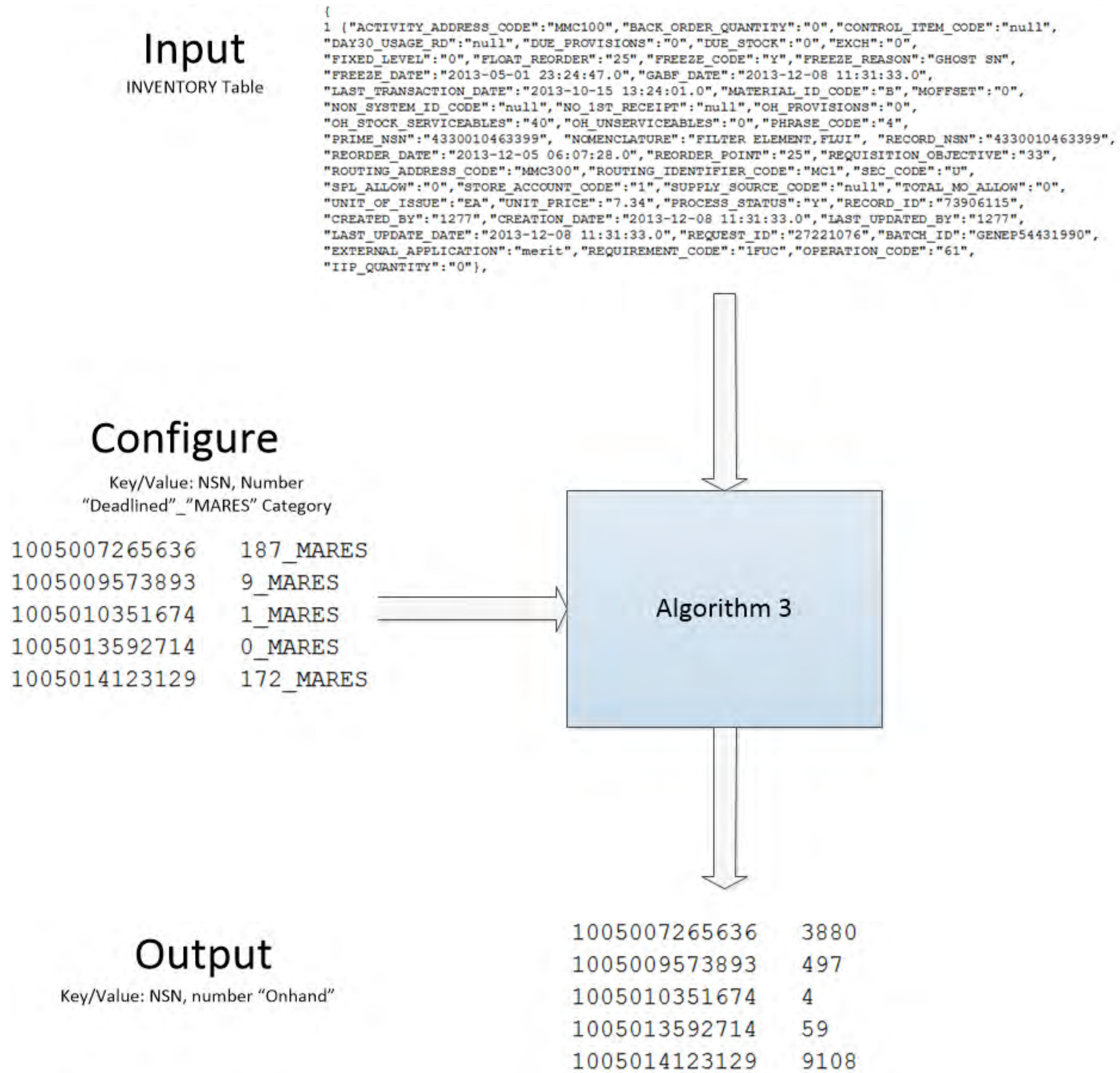


Figure 4.20: Alpha: Third Algorithm

ability to use multiple inputs to perform reduce side joins. We purposely showed the map side joins because we felt like it was easier to demonstrate the data flow with map side joins. However, the number of algorithms can be reduced by using reduce side joins. There is also a small speed up that can be achieved on the reduce side join, but it is not significant because all of the data still needs to be scanned the same amount.

```
1005007265636   3880
1005009573893   497
1005010351674   4
1005013592714   59
1005014123129   9108
```

Figure 4.21: Alpha Output from the Third Algorithm

Table 4.3: Sample of HDFS_ALPHA Table

NSN	NUMBER_DEADLINED	MARES_CATEGORY	QUANTITY_ONHAND	PERCENTAGE
3895014538573	4	MARES	24	16.67
3895015390585	1	MARES	22	4.55
3895015508369	0	MARES	25	0.00
3895015733847	1	MARES	1	100.00
3930014783519	54	MARES	389	13.88
3930014862151	1	MARES	6	16.67
3930015080886	58	MARES	541	10.72
3930015227364	6	MARES	106	5.66
3930015330855	22	MARES	187	11.76
3930015735873	2	MARES	5	40.00

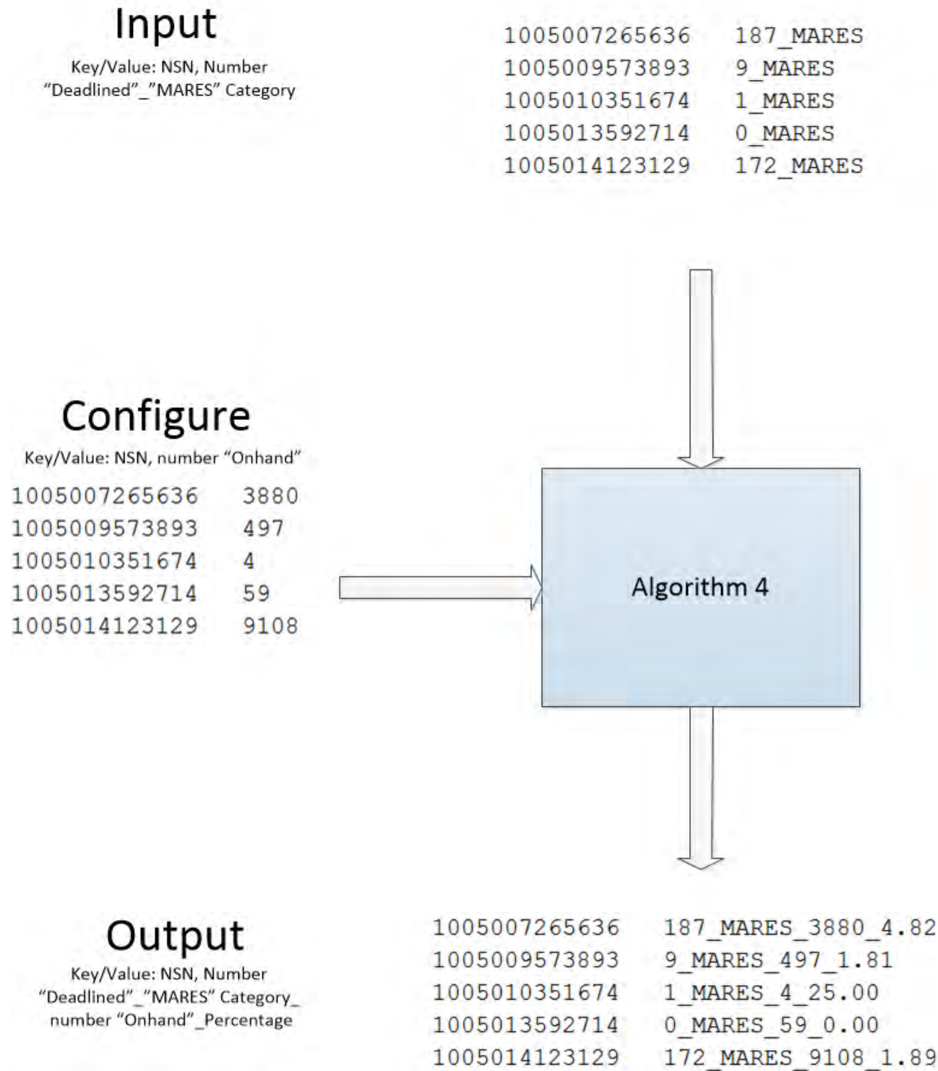


Figure 4.22: Alpha: Fourth Algorithm

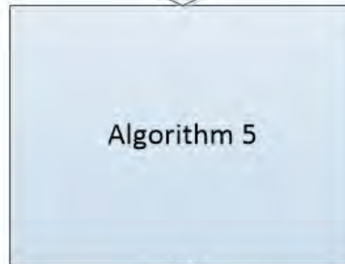
```
1005007265636   187_MARES_3880_4.82
1005009573893   9_MARES_497_1.81
1005010351674   1_MARES_4_25.00
1005013592714   0_MARES_59_0.00
1005014123129   172_MARES_9108_1.89
```

Figure 4.23: Alpha Output from the Fourth Algorithm

Input

Key/Value: NSN, Number "Deadlined"_"MARES" Category_ number "Onhand"_Percentage

```
1005007265636    187_MARES_3880_4.82
1005009573893    9_MARES_497_1.81
1005010351674    1_MARES_4_25.00
1005013592714    0_MARES_59_0.00
1005014123129    172_MARES_9108_1.89
```



Output

Key/Value: NSN, SQL Insert Statement

```
1005007265636    INSERT INTO HDFS_ALPHA(NSN, NUMBER_DEADLINED ...)
1005009573893    INSERT INTO HDFS_ALPHA(NSN, NUMBER_DEADLINED ...)
1005010351674    INSERT INTO HDFS_ALPHA(NSN, NUMBER_DEADLINED ...)
1005013592714    INSERT INTO HDFS_ALPHA(NSN, NUMBER_DEADLINED ...)
1005014123129    INSERT INTO HDFS_ALPHA(NSN, NUMBER_DEADLINED ...)
```

Figure 4.24: Alpha: Fifth Algorithm

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Conclusion

The amount of data that is collected and stored continues to increase everyday. The International Data Corporation (IDC) estimates that by the end of 2013 data stored data will be 2.7 zettabyte (ZB)s, that is a forty-eight percent increase from 2011 [19]. The USMC is no different than the commercial world. The size of data stored in USMC databases is growing, with the current size of the GCSS-MC database at six TBs. There is a need to find a solution to manage data storage increases in the USMC. This thesis demonstrated that big data analytics could be added to the current GCSS-MC architecture to address the issue of giving the USMC the power of using all of their data in developing analytics. The remainder of this chapter further explains how the research questions have been answered and covers future work.

5.1 The Outcome

The power of processing GCSS-MC data in Hadoop is promising. The thesis shows examples of the analytics that can be run in the Hadoop ecosystem on the GCSS-MC data. This work shows promise that a Hadoop cluster can handle the analytics that are needed now and is flexible enough to allow for the programming on additional analytical needs. With continued efforts and exploration, as previously mentioned, the power of big data analytics could be at our fingertips providing function and simplicity to a complicated large data set.

The research questions posed in Chapter 1 were the guide to the overall proof of concept behind adding big data analytics to GCSS-MC. We determined that a big data element can be added to the GCSS-MC system and that it can be used to provide data analytics within the GCSS-MC system. After completing the research and examining the results we found that some of the questions were broad or vague. We decided to pursue this research using HDFS and as small of a software footprint as possible. This thesis shows that concept of adding Hadoop to the GCSS-MC system is achievable. However, more work is needed to show how Hadoop could be integrated into a system that more closely resembles a production GCSS-MC system. The rest of this section will reiterate the research questions and indicate where the details of the research can be found within the thesis.

The first research question is: What would an architecture look like that adds a big data element to GCSS-MC? This question is very broad and can be approached several ways. In this thesis we decided to use HDFS as the big data element to add to the GCSS-MC system. Furthermore, we built a HDFS cluster and showed how it could be used as a benefit to the GCSS-MC system. Chapter three and four explain these results in detail.

The second research question is: If an architecture can be developed, what modifications would the GCSS-MC architecture need? As discussed, in detail in chapter three. We are using a sample of the GCSS-MC database and demonstrating cluster interaction with that database though IP connectivity. This abstraction of the real GCSS-MC system worked very well in this thesis. We found that we were able to integrate the HDFS solution rather seamlessly into our sample database and experiment architecture. In order to fully prove that this approach will work, it needs to be further tested on an actual implementation of the GCSS-MC system.

The third research question is: How can the data contained in GCSS-MC be imported into HDFS? In this thesis we used a JDBC to connect to the sample GCSS-MC database to parse the data in JSON format. Then the data is imported into HDFS with a bash script. Chapter four explains in more detail the process that was used to ultimately get the GCSS-MC data into the HDFS ecosystem.

The fourth research question is: What type of analytics can Hadoop provide for GCSS-MC data? This thesis explores two explanations of how to provide analytics on GCSS-MC data. Those example are by no means the only analytics that HDFS can provide. They are merely representative of what HDFS can provide. More specifically, the Alpha program is an example of a analytic the USMC is asking to be completed on the GCSS-MC data. Chapter four explains the code of both programs in great detail.

The fifth and final research question is: How will the data get back to the GCSS-MC database? We choose to write back to the GCSS-MC database in the MapReduce code. We achieved this by using a JDBC call within the map phase of the MapReduce code. Although this is not the only way to achieve writing data to a database from Hadoop, we felt it was the best way to achieve the data write back functionality. Chapter four discusses this code.

5.2 Future Work

This thesis demonstrates the potential of using a Hadoop cluster as a big data element in the GCSS-MC system. There are a few things that could be further examined to enhance a big data element of the GCSS-MC system. This section will focus on three additional research areas that could be extended from this thesis.

The first area of research that can be extended from this thesis is moving the cluster to a larger environment. We set up, the cluster on two machines and all of the nodes are virtualized. In order to better represent what would be seen in a GCSS-MC production environment, a cluster with greater than twenty nodes should be used. Additionally, the security of the cluster and the interconnectivity between the cluster and the GCSS-MC should be examined.

The next area of research could be a comparison between SQL and Hadoop analytics. For instance, the Alpha program is derived from an analytic the USMC requested in the GCSS-MC system. The Alpha program could run on a cluster and the SQL version could be executed on a database and the runtime, compute power, and memory usage could be compared between the cluster and database. Furthermore, there are four more analytics that can be written in MapReduce to allow for a more complete comparison.

The final extension of research to the thesis could be the addition of Hadoop big data tools. Hadoop offers several tools that help in the processing, analyzing, and importing/exporting data. For instance, some effort could be placed on comparing import/export tools against one another to discover which tools are best for the GCSS-MC data. Hadoop also offers a tool called Hive. Hive allows data to be loaded into Hadoop and SQL-like queries can be run on the data. Some effort might be placed on examining performance metrics between running a MapReduce analytic versus running the same analytic in Hive.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A:
Hadoop Testing on Single and Two Node Clusters

The Hadoop Distributed File System (HDFS) is a cluster computing technology that uses a parallel computing architecture to provide fast computing and redundant storage. The HDFS system is an open source project from Apache that is based off of Jeffrey Dean and Sanjay Ghemawat's MapReduce paper [6]. The intent of HDFS is to bring a reliable fault tolerant parallel architecture to the open source community.

The main idea behind the parallel computing architecture is to build an architecture that will allow faster computing than a single CPU will allow. Even with Moore's Law producing computing power that doubles every twelve to eighteen months there is still a need for faster computing to deal with large amounts of data. Systems like HDFS hope to answer the demand of processing large amounts of data quickly over a distributed environment. HDFS allows processing of large amounts of data using a cluster with 1000s of nodes (i.e., networked computers).

This is not to say that HDFS is a panacea and the answer for all problems. For instance some problems do not lend themselves to the MapReduce paradigm, that is to say that some problems will not see a speed up when computed in a cluster environment. The problems that lend themselves well to the MapReduce paradigm are those that can be split into small chunks and computed without changing the final outcome. The canonical example of this is the WordCount Program. This program takes multiple files as an input and computes the number of occurrence of each word in all of the input files. This can be illustrated by running the computation of the word count on each individual file on a separate node and then combining that result. This can be achieved significantly quicker than it would be to allow one computer to process each file individually. However, you can see that the additional step of combining the intermediate files adds latency that would not exists if done by a single computer. That is considered to be overhead in the MapReduce algorithm. There are other sources of overhead that need to be considered as well (e.g., network latency, data locality, etc).

The purpose of this experiment is to analyze the speed up of a HDFS cluster. This will be tested running several benchmarks on two different clusters. The first cluster will be a single node cluster and the second cluster will be a two node cluster. Every effort will be kept to control the environment to show that any speed up or slowdown will be contributed to the HDFS architecture and not errors in the experiment.

A.1 Hypothesis

The experiment will examine several benchmarks to produce a speed up factor over each benchmark. The hypothesis is that the two node cluster will perform better than the single node cluster. The two node cluster will perform around 1.75 times better than the single node cluster due to the overhead required by HDFS in the two node cluster to distribute the work between the nodes.

A.2 Experiment Architecture

The experiment utilized a virtual environment to set up the two clusters. The settings for each node were kept the identical in order to minimize any variance in the experiment setup. Each node was built in Oracle VirtualBox version 4.3.2 using a Linux Ubuntu release, see Table A.1 for additional details. Each node was built separately and configured separately in VirtualBox. HDFS1 was set up and configured to be the single node cluster. HDFS2 and HDFS3 were set up and configured to be a two node cluster.

Table A.1: Initial Node Settings

Node Name	HDFS1	HDFS2	HDFS3
OS	Ubuntu 13.10 (64Bit)	Ubuntu 13.10 (64Bit)	Ubuntu 13.10 (64Bit)
RAM Size	4096 MB	4096 MB	4096 MB
HD Size	100.77 GB	100.77 GB	100.77 GB

In the configuration of the node set up there is a replication factor that is set. This factor is how many times the data is replicated across the nodes in a cluster. This factor was overlooked when the initial hypothesis was considered. When a node has to write the same data more than once the time to write is going to take longer. However, if you reduce that factor to one on a two node cluster than the read time takes longer, because the systems

experiences overhead when a node has to pass the data to the other node to read. The results can be found in Section A.4. The experiment architecture (see Figure A.1) was set up and tested to ensure the basic WordCount program would run. After the initial testing was run and confirmed to be accurate the nodes were ready to begin testing.

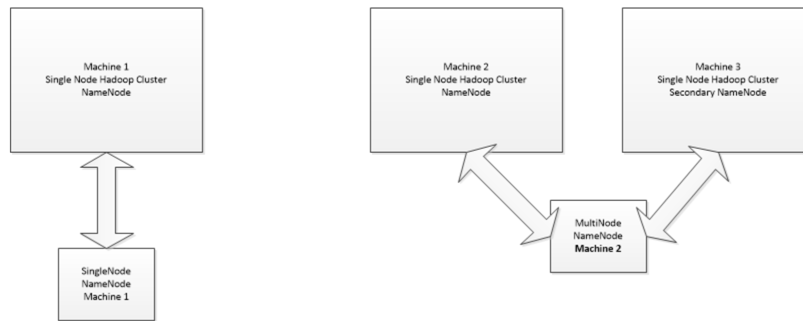


Figure A.1: Experiment Architecture

A.3 Experiments Run

The tests that were run in this experiment were a write test, a read test, and a TeraSort test. These benchmarks are all available as part of the HDFS release. The tests were configured to maximize the effectiveness of the experimental architecture. The setup includes single-node and two-node configurations. When configuring our two-node system the replication factor was accidently set to two. This meant that when we are writing to HDFS the cluster has to write the data twice. We recognized this late into our experiment, but still had time to perform additional tests with a replication factor of one. Each configuration underwent twenty Read, Write, and TeraSort jobs.

The Write test would generate ten files that are 1GB each, thus putting 10GB on the cluster. In between read jobs, the cluster would be cleaned to prohibit the drive from recognizing redundancy and refusing to write for subsequent tests. The Read test will read ten files of 1GB each. We simply do not delete the files from the last run of the write test. Finally we subject each configuration to a TeraSort test. This test will first generate 10,000,000 rows of one-hundred byte data. In total this is 1GB of data to sort. After each run the output files will be removed.

Each test would be ran twenty times against each configuration, we collected and summa-

rized the data below in the following charts.

A.4 Results

As we mentioned previously, write testing on the two-node setup takes twenty percent longer than on a single node configuration (see Figure A.2) when using a replication factor of two, this is despite the fact that it is using twice the system resources. This can be attributed to the fact that our test VMs are using the same underlying hard-drives. If we had a true set of servers, this result set would likely look much different. Another reason for the slowdown is because the HDFS has to write the data twice for redundancy. A standard HDFS cluster uses triple-redundancy, but that overhead is shared over dozens of servers.

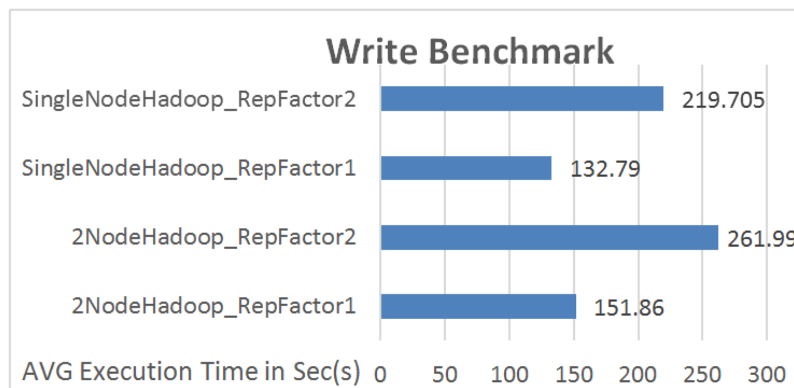


Figure A.2: Write Benchmark Results

When we get to the Read tests (Figure A.2), our results are more as we expected. When reading in a DFS, if the node requesting data does not have it stored locally, it results in a performance impact. Thus our two-node setups lagged twenty percent and sixty-six percent respectively, behind the single node configuration. Interesting from an outside standpoint, the two-node double replication read ran quicker than a single replication read. This is a strength of a DFS, since it will front-load the processing time to write data to multiple nodes. The result is much faster read times; since the data is in multiple places for nodes to request it and in our simulation, that meant both nodes had a copy of the data to read.

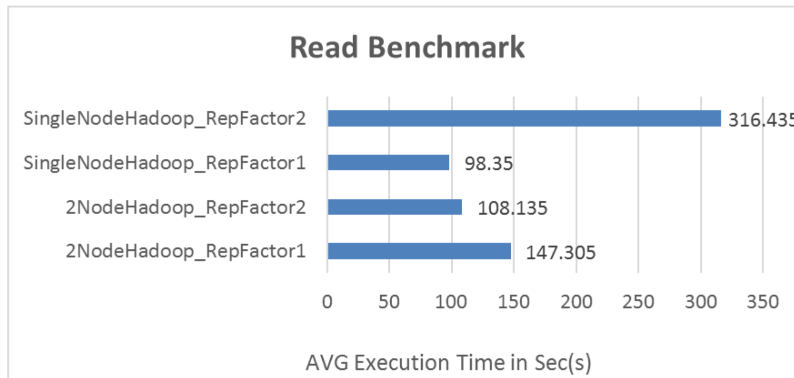


Figure A.3: Read Benchmark Results

In our TeraSort tests (Figure A.4), we found that utilizing a second node provided a clear improvement in average time. Both the single and dual replication two node configurations completed nearly sixty percent faster than any single node configuration.

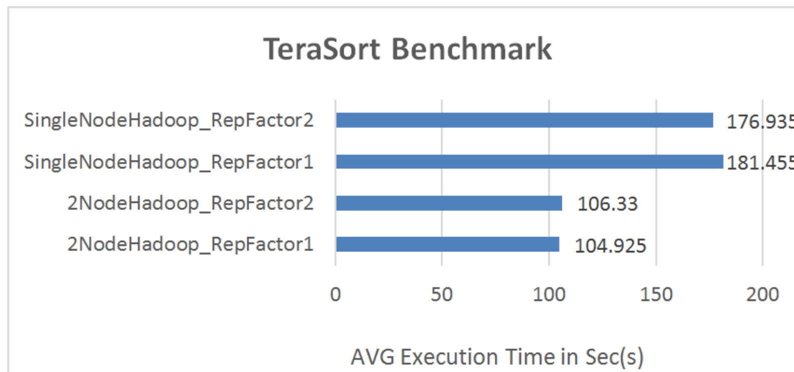


Figure A.4: TeraSort Benchmark Results

A.5 Conclusion

We concluded that even though we were able to see performance improvements with the TeraSort and Read benchmark our underlying architecture prevented us from seeing that same sort of improvements in the Write tests. It is our belief that the main reason the write tests do not show a performance increase is caused by the replication factor. Exacerbating the issue was the fact the tests were performed in a virtual environment that has to access

the same hard drive (HD) to write. In other words, the write operation can not be done in parallel.

In order to try and limit these side effects we attempted to alter the replication factors to try and to gain an equal comparison. This effort led to more data that confirmed some of our thoughts. At first, we forced the two node cluster to a replication factor of one. So with the replication factor of one being the same between the single node and two node then the write should perform the same or better on the two node system due to the bottleneck of the HD speed. However, what we saw was the two node cluster took longer to write the same amount of data. After studying the HDFS architecture we came to the conclusion that the replication factor sets how many times the data is written, but not where the data is written. So, on the two node cluster the master node (NameNode) is going to try to balance the data on both nodes and in doing so is going to experience network overhead which is what causes the a delay. Additionally, because the nodes are both trying to write at the same time to the same physical HD they are going to experience a wait for the access to the physical drive. With the physical HD bottleneck and some network overhead it is logical to think that it can account for the additional 18ms it takes to write the data on the two node cluster.

Moreover, we set both the two node cluster and the single node cluster to a replication factor of two. This seems unrealistic on a single node system because the single node is just going to write the same data twice on the same HD. Nevertheless, the tests were run in an attempt to gain a fair level of comparison. What we found was the single node performed better than the two node cluster on the write test. Again we can contribute this to the fact that the two node system is spitting the work up between the nodes causing some network overhead and when the actual write happens it is writing to the same physical HD causes a slowdown as the process has to wait for access to the HD. With the HD factor and the network overhead the slowdown of 41ms seems logical.

After performing all of these tests and studying the HDFS architecture we belief on order to make a fair comparison of speed up would be to perform the test on physical cluster rather than virtual instances of clusters. The test would also be more effective if we could run the tests on say a three node cluster verse a six node cluster. These are still both considered small clusters, but they should be big enough to use a default replication factor of 3 and see the performance increase at a fair level of comparison.

APPENDIX B:
How to Setup a Single Node Hadoop Cluster

This tutorial will guide you through installing a single node Hadoop cluster. This tutorial was built using Tom White's *Hadoop: The Definitive Guide* [7] and Michael Noll's *Hadoop Tutorials* [17]. A single node cluster is a great way to start working with the Hadoop Distributed File System and the MapReduce paradigm. Once you have installed and tested the cluster, all done in this tutorial, you will have your environment set up to start testing MapReduce code.

Notes:

- This tutorial assumes that you are installing the node on a debian based linux platform.
- This can be done on actual hardware or on a virtual machine.
- It is recommended that you have at least 4GB of RAM and 100GB of free hard disk space.

1. open up a terminal
2. Update current packages
   (a) `sudo apt-get update`
3. Install Java
   (a) `sudo apt-get install openjdk-7-jdk`
4. Verify the install
   (a) `java -version`
5. Install nano
   (a) `sudo apt-get install nano`
   - this is a terminal text editor. You may skip this if you choose to use another terminal text editor (i.e. VI or emacs)
6. Hadoop uses ssh to talk from the local machine to the namenode. We have to configure ssh for that.
   (a) `ssh-keygen -t rsa -P ""`

- save to default file

(b) if you do not have a ssh server install one

- `sudo apt-get install openssh-server`

7. add the key we just created to the authorized key file

   (a) `cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys`

8. test the key

   (a) `shh localhost`

   - should give you a message that localhost has been added to list of known hosts

9. Hadoop and Ubuntu have a conflict with IPV6. The workaround we will use is to disable IPV6

   (a) `sudo nano /etc/sysctl.conf`

   - add these lines to the end of the file
     - `# disable ipv6`
     - `net.ipv6.conf.all.disable_ipv6 = 1`
     - `net.ipv6.conf.default.disable_ipv6 = 1`
     - `net.ipv6.conf.lo.disable_ipv6 = 1`

10. In order for the changes to take effect you have to reboot the machine

    (a) `sudo reboot`

11. once the machine has rebooted, open a terminal

12. check to see if the changes took place

    (a) `cat /proc/sys/net/ipv6/conf/all/disable_ipv6`

    - you want to see a return of 1

13. change directory

    (a) `cd /usr/local`

14. Download Hadoop

    (a) `sudo wget http://download.nextag.com/apache/hadoop/common/hadoop-1.2.1/hadoop-15`

15. extract the tarball

    (a) `sudo tar xzf hadoop-1.2.1.tar.gz`

16. move the folder and change permissions

    (a) `sudo mv hadoop-1.2.1 hadoop`

    (b) `sudo chown -R <your username>:<your group> hadoop`

17. update the .bashrc file for the hdfsuser
    (a) sudo nano /home/<your username>/.bashrc
        - add the following lines to the file
            - export HADOOP_HOME=/usr/local/hadoop
            - export JAVA_HOME=/usr/lib/jvm/java-7-openjdk-amd64
            - unalias fs &> /dev/null
            - alias fs="hadoop fs"
            - unalias hls &> /dev/null
            - alias hls="fs -ls"
            - lzohead () {
            - hadoop fs -cat $1 lzop -dc | head -1000 | less|
            - }
            - export PATH=$PATH:$HADOOP_HOME/bin
18. create the directory that Hadoop will use to store files
    (a) sudo mkdir -p /app/hadoop/tmp
    (b) sudo chown <your username>:<your group> /app/hadoop/tmp
19. Edit configurations files for use in your environment and point to the directory we just created
    (a) change the Hadoop-env.sh file for the java jdk you installed
        - sudo nano /usr/local/hadoop/conf/hadoop-env.sh
            - uncomment the export JAVA_HOME and give it correct path
                * export JAVA_HOME=/usr/lib/jvm/java-7-openjdk-amd64
20. change the core-site xml file
    (a) sudo nano /usr/local/hadoop/conf/core-site.xml
        - add the following in between the configuration tags
            - <property>
            - <name>hadoop.tmp.dir</name>
            - <value>/app/hadoop/tmp</value>
            - </property>
            - <property>
            - <name>fs.default.name</name>
            - <value>hdfs://localhost:54310</value>

- `</property>`

21. change the mapred-site.xml file
    (a) `sudo nano /usr/local/hadoop/conf/mapred-site.xml`
        - add the following in between the configuration tags
            - `<property>`
            - `<name>mapred.job.tracker</name>`
            - `<value>localhost:54311</value>`
            - `</property>`
22. change the hdfs-site.xml
    (a) `sudo nano /usr/local/hadoop/conf/hdfs-site.xml`
        - add the following in between the configuration tags
            - `<property>`
            - `<name>dfs.replication</name>`
            - `<value>1</value>`
            - `</property>`
23. Format the HDFS file system and the name node
    (a) `/usr/local/hadoop/bin/hadoop namenode -format`
        - you should see some output during creation
24. start your cluster
    (a) `cd /usr/local/hadoop`
    (b) `bin/start-all.sh`
25. run jps to see that all of the nodes started
    (a) `jps`
26. Download book for wordcount
    (a) make directory on the local file system
        - `mkdir /tmp/book`
    (b) cd to that directory
        - `cd /tmp/book`
    (c) download book
        - `wget http://www.textfiles.com/games/abc.txt`
    (d) copy files from local file system to hdfs
        - `cd /usr/local/hadoop`

- bin/hadoop dfs -copyFromLocal /tmp/book /user/book

(e) check to see if it is there

- bin/hadoop dfs -ls /user/
- bin/hadoop dfs -ls /user/book

27. run pre-compiled wordcount

    (a) bin/hadoop jar hadoop*examples*.jar wordcount /user/book /user/book-output

28. make sure the output is there and look at it

    (a) bin/hadoop dfs -ls /user/

    (b) bin/hadoop dfs -ls /user/book-output

    (c) bin/hadoop dfs -cat /user/book-output/part-r-00000

29. create a local directory and move files there

    (a) mkdir /tmp/book-output

    (b) bin/hadoop dfs -getmerge /user/hdfsuser/books-output-3 /tmp/book_output

30. stop cluster

    (a) bin/stop-all.sh

This completes the tutorial. You are now reading to begin learning how to use the Map Reduce paradigm and writing your own programs. As a place to start, I recommend modifying the Wordcount example on the Hadoop website.

THIS PAGE INTENTIONALLY LEFT BLANK

References

[1] J. Bamford, "The NSA is building the country's biggest spy center," *Wired*, vol. 15, 2012. [Online]. Available: http://www.wired.com/2012/03/ff_nsadatacenter/all/

[2] A. A. Khan, *DOD business transformation: Improved management oversight of business system modernization efforts needed*. U.S. Govermennt Accountability Office: Washington DC, REP. GAO 1153, 2010.

[3] R. C. Hite, M. Anatalio, H. Brumm, N. Doherty, C. Dottermusch, N. Glover, M. Hassan, M. Holland, E. Iczkovitz, and A. Le, *DOD BUSINESS SYSTEMS MODERNIZATION: Key Marine Corps System Acquisition Needs to be Better Justified, Defined, and Managed*. U.S. Govermennt Accountability Office: Washington DC, REP. GAO 08822, 2008.

[4] A. Dwyer. (2012). GCSS-MC industry day overview and program status v5 Global Combat Support System Marine Corps (GCSS-MC)program overview and status. [Online]. Available: http://www.defenseinnovationmarketplace.mil/resources/GlobalCombatSupportSystem.pdf

[5] T. Kalil. (2012, March 29). Big data is a big deal. [Online]. Available: http://www.whitehouse.gov/blog/2012/03/29/big-data-big-deal

[6] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[7] T. White, *Hadoop: The Definitive Guide*. Sebastopol, CA: O'Reilly Media, 2009.

[8] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. Incline Village, NV: IEEE, 2010, pp. 1–10.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[10] R. E. Bryant, "The case for disc," Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-CS-07-128, Nov 2007.

[11] T. A. S. Foundation. (2014). Hadoop. [Online]. Available: http://hadoop.apache.org/

[12] T. A. S. Foundation. (2014). Jobconf(apache hadoop. [Online]. Available: http://hadoop.apache.org/docs/r2.3.0/api/org/apache/hadoop/mapred/JobConf.html

[13] A. Chaudhary and P. Singh, “Big data–importance of Hadoop distributed filesystem,” in *International Journal of Scientific and Engineering Research*, vol. 4, no. 11, Nov 2013.

[14] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, “Mapreduce online.” in *NSDI*, vol. 10, no. 4, 2010, p. 20.

[15] Department of Defense. (2014). DOD open government. [Online]. Available: http://open.defense.gov/Data.aspx

[16] L. Lei, “Towards a high performance virtual Hadoop cluster,” *JCIT*, vol. 7, no. 6, pp. 292–303, 2012.

[17] M. G. Noll. (2014). Tutorials. [Online]. Available: http://www.michael-noll.com/tutorials/

[18] D. Borthakur. (2010). HDFS: Facebook has the world’s largest Hadoop cluster! [Online]. Available: http://hadoopblog.blogspot.com/2010/05/facebook-has-worlds-largest-hadoop.html

[19] P. Doscher. (2013). Searching for dark data. [Online]. Available: http://siliconangle.com/blog/2013/02/11/searching-for-dark-data/

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California