UNLV University Libraries
University of Nevada, Las Vegas

UNLV Theses, Dissertations, Professional Papers, and Capstones

May 2018

Deep Data Locality on Apache Hadoop

Sungchul Lee
lsungchul@gmail.com

Follow this and additional works at: https://digitalscholarship.unlv.edu/thesesdissertations

Part of the Computer Sciences Commons

Repository Citation
Lee, Sungchul, "Deep Data Locality on Apache Hadoop" (2018). *UNLV Theses, Dissertations, Professional Papers, and Capstones*. 3280.
https://digitalscholarship.unlv.edu/thesesdissertations/3280

This Dissertation is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Theses, Dissertations, Professional Papers, and Capstones by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.

DEEP DATA LOCALITY ON APACHE HADOOP

By

Sungchul Lee

Bachelor of Science in Computer Engineering
Konkuk University, South Korea
2009

Master of Science in Computer Science
University of Nevada, Las Vegas
2012

A dissertation submitted in partial fulfillment
of the requirements for the

Doctor of Philosophy in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
May 2018

Copyright 2018, by Sungchul Lee

All Rights Reserved

Dissertation Approval

The Graduate College
The University of Nevada, Las Vegas

May 10, 2018

This dissertation prepared by

Sungchul Lee

entitled

Deep Data Locality on Apache Hadoop

is approved in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science
Department of Computer Science

Dr. Yoohwan Kim, Ph.D
Examination Committee Chair

Dr. Ju-Yeon Jo, Ph.D
Examination Committee Co-Chair

Dr. Laxmi Gewali, Ph.D
Examination Committee Member

Dr. Justin Zhan, Ph.D
Examination Committee Member

Dr. Haroon Stephen, Ph.D
Graduate College Faculty Representative

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Interim Dean

ABSTRACT

Deep Data Locality on Apache Hadoop

By

Sungchul Lee

Dr. Yoohwan Kim, Examination Committee Co-Chair
Dr. Ju-Yeon Jo, Examination Committee Co-Chair
School of Computer Science
University of Nevada, Las Vegas

The amount of data being collected in various areas such as social media, network, scientific instrument, mobile devices, and sensors is growing continuously, and the technology to process them is also advancing rapidly. One of the fundamental technologies to process big data is Apache Hadoop that has been adopted by many commercial products, such as InfoSphere by IBM, or Spark by Cloudera. MapReduce on Hadoop has been widely used in many data science applications. As a dominant big data processing platform, the performance of MapReduce on Hadoop system has a significant impact on the big data processing capability across multiple industries. Most of the research for improving the speed of big data analysis has been on Hadoop modules such as Hadoop common, Hadoop Distribute File System (HDFS), Hadoop Yet Another Resource Negotiator (YARN) and Hadoop MapReduce. In this research, we focused on data locality on HDFS to improve the performance of MapReduce. To reduce the amount of data transfer, MapReduce has been utilizing data locality. However, even though the majority of the processing cost occurs in the later stages, data locality has been utilized only in the early stages, which we call Shallow Data Locality (SDL). As a result, the benefit of data locality has not been fully realized. We have explored a new concept called Deep Data Locality (DDL) where the data

is pre-arranged to maximize the locality in the later stages. Specifically, we introduce two implementation methods of the DDL, i.e., block-based DDL and key-based DDL.

In block-based DDL, the data blocks are pre-arranged to reduce the block copying time in two ways. First the RLM blocks are eliminated. Under the conventional default block placement policy (DBPP), data blocks are randomly placed on any available slave nodes, requiring a copy of RLM (Rack-Local Map) blocks. In block-based DDL, blocks are placed to avoid RLMs to reduce the block copy time. Second, block-based DDL concentrates the blocks in a smaller number of nodes and reduces the data transfer time among them. We analyzed the block distribution status with the customer review data from TripAdvisor and measured the performances with Terasort Benchmark. Our test result shows that the execution times of Map and Shuffle have been improved by up to 25% and 31% respectively.

In key-based DDL, the input data is divided into several blocks and stored in HDFS before going into the Map stage. In comparison with conventional blocks that have random keys, our blocks have a unique key. This requires a pre-sorting of the key-value pairs, which can be done during ETL process. This eliminates some data movements in map, shuffle, and reduce stages, and thereby improves the performance. In our experiments, MapReduce with key-based DDL performed 21.9% faster than default MapReduce and 13.3% faster than MapReduce with block-based DDL. Additionally, key-based DDL can be combined with other methods to further improve the performance. When key-based DDL and block-based DDL are combined, the Hadoop performance went up by 34.4%.

In this research, we developed the MapReduce workflow models with a novel computational model. We developed a numerical simulator that integrates the computational models. The model faithfully predicts the Hadoop performance under various conditions.

ACKNOWLEDGMENTS

I would like to give my most respect and gratitude to my dissertation advisors, Dr. Yoohwan Kim and Dr. Ju-Yeon Jo, for chairing my committee and advising me throughout my dissertation work. His support, enthusiasm, and most importantly, his confidence in my abilities, have helped me greatly throughout my graduate study. They have constantly enlightened me with sincere care and guidance not only for my academics but also personal life. I am truly honored for the opportunity of interacting with him throughout the program, especially on completing the dissertation. I am also very grateful to Dr. Laxmi Gewali, Dr. Justin Zhan and Dr. Haroon Stephen for having supported me, throughout my academic years.

DISCLAIMER

This material is based upon the work supported by the National Science Foundation under grant number IIA-1301726. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGEMENTS v

DISCLAIMER vi

TABLE OF CONTENTS vii

LIST OF TABLES x

LIST OF FIGURES xii

CHAPTER 1. INTRODUCTION 1

1.1 Motivation 1

1.2 Hadoop Performance Research 2

1.3 Overview of Our Works 5

CHAPTER 2. HADOOP SYSTEM 7

2.1 Master-Slave Architecture 8

2.2 Hadoop Distributed File System 11

2.2.1 Block System in HDFS 13

2.2.2 Block Replication 14

2.3 Basic Architecture of MapReduce 15

2.3.1 MapReduce on Hadoop Version 2.0 18

CHAPTER 3. DATA LOCALITY 22

3.1 Basic Concept of Data Locality 23

3.2 Example of Data Locality with Test 25

3.3 Basic Scheme for Data Locality 28

CHAPTER 4. RELATED WORK OF DATA LOCALITY RESEARCH 30
4.1 Data Locality in Map 30
4.2 Data Locality in Shuffle 34
4.3 Data Locality in Reduce 36
4.4 Analyzing Previous Data Locality Research 37
4.5 Performance Testing 41
4.5.1 Terasort Benchmark 41
4.5.2 WordCount 43
4.6 Limitation of Shallow Data Locality 44
CHAPTER 5. MAPREDUCE COMPUTATIONAL MODEL 47
5.1 Simple Computational Model 47
5.2 Advanced Analyzing Hadoop Performance Model 50
5.2.1 First Stage (T_1) 51
5.2.2 Second Stage (T_2) 53
5.2.3 Total Hadoop Processing Time (T) 57
CHAPTER 6. DEEP DATA LOCALITY 60
6.1 The Concept of Deep Data Locality 61
6.2 Block-based DDL 61
6.2.1 LNBPP 62
6.2.2 Performance Analysis of LNBPP 66
6.3 Key-based Deep Data Locality 71
6.3.1 DDL-Aware ETL 73
6.3.2 Performance Analysis of Key-based DDL 75

6.3.3 Limitations of Key-based DDL 79
CHAPTER 7. CONCLUSIONS AND FUTURE WORK 81
APPENDIX A Data Locality Simulator 83
APPENDIX B Sample Test Sheet 90
APPENDIX C Hadoop Configuration 97
APPENDIX D Hardware Implement 101
REFERENCES 102
CURRICULUM VITAE 111

LIST OF TABLES

2-1 Hadoop version 1 VS. Hadoop version 2 10

3-1 Block location in slave nodes 26

3-2 Effect of RLM on T_M and T_P on DBPP 27

3-3 Allocated slave nodes to process MapReduce on HDFS 28

4-1 Time line of data locality research 37

4-2 Category of application type and highest efficiency research 39

5-1 Constants 50

5-2 Time Variables 50

5-3 Other Variables 51

6-1 Terasort with DBPP and LNBPP 67

6-2 Performance improvement by Key-based DDL 78

6-3 Performance improvement by Block & Key DDL 78

6-4 Comparison between DDL-aware ETL and Big-ETL 79

B-1 Default MapReduce 90

B-2 DDL-Aware ETL 91

B-3 Block-based DDL without ETL 92

B-4 Block-based DDL with DDL-Aware ETL 93

B-5 Key-based DDL 95

B-6 Key-Block based DDL 96

C-1 hdfs-site.xml 100

C-2 mapred-site.xml 100

C-3 yarn-site.xml 100

LIST OF FIGURES

1-1 Composition of Hadoop System with Physical Hardware and 3rd Party Software 3

1-2 Moving Code to Slave Nodes 4

2-1 Hadoop VM daemon 9

2-2 Basic Architecture of HDFS 12

2-3 Hadoop Split Input Data into Several Blocks 13

2-4 Basic Replica Placement Policy on HDFS 14

2-5 Basic Architecture of MapReduce in Hadoop Version 1.X 16

2-6 Hadoop Shuffling.. 17

2-7 The Workflow of MapReduce on Hadoop Version 2.0 18

3-1 Data Locality in HDFS 24

3-2 Effect of RLM on T_M and T_P on DBPP 27

4-1 Map Stage 31

4-2 Clustering Locality Research 38

4-3 Process Time of Map in Terasort 41

4-4 Process Time of Shuffle 42

4-5 WordCount Process Time with Default and Data Placement 43

4-6 Processing Times of Each Step of MapReduce... 45

4-7 Changing Data Locality between Map and Reduce Step 46

5-1 Time Variables for Each Stage... 51

5-2 Map Function time.. 52

5-3 Processing Time of First Stage on Mapper (i)... 53

5-4 First Stage Calculation Graph 54
5-5 Data Locality in Second Stage 54
5-6 Transfer Time (T_T) to reducer (j) 55
5-7 The Process of the Second State (T_2) on the Reducer (j) 56
5-8 Second Stage Calculation Graph (T_2) 57
5-9 Total Hadoop Processing Time Calculation Graph 58
6-1 Decreased Shuffle Time by LNBPP 63
6-2 Block Locations in MapReduce (a) DBPP (b) LNBPP 64
6-3 (a) Map Performance, (b) Shuffle Performance, (c) Total Performance 67
6-4 An example of mappers' processing times (ms) (a) DBPP (b) LNBPP 69
6-5 (a) Default M/R VS. (b) M/R with Key-Based DDL 71
6-6 Traditional ETL 72
6-7 DDL-Aware ETL using Big-ETL 73
6-8. Shuffle Stage (a) Default block (b) Pre-partitioned block by DDL-Aware ETL 74
6-9. Performance Test of Map 76
6-10 Performance Test of Shuffle 77
6-11. Performance Test of MapReduce 78
A-1. DDL Simulator 83
A-2. Block Allocations in the Simulator 84
A-3. MapProc 85
A-4. ShufflePro and ReducerPro (a) Analysis of the Shuffle (b) Analysis of the Reduce 86
A-5. Total Processing Time of MapReduce with Each Stage 88
C-1. Memory control in Hadoop 97

CHAPTER 1. INTRODUCTION

1.1 Motivation

This chapter introduces the big data trends and the principles of the Apache Hadoop. It describes how Hadoop resolves the network congestion to handle the big data traffic load. There are three methods to improve the performance of Hadoop, that is, 3rd party software on YARN, hardware, and data locality, which is described in this chapter. We also briefly introduce the data locality concept and the limitations of traditional approaches on data locality in Hadoop.

The amount of data being collected in various areas such as social media, network, scientific instrument, mobile devices, and sensors is growing continuously, and the technology to process them is also advancing rapidly. Every year, the data production is growing faster. 2.5 quintillion bytes of data are generated [113]. It is bigger than the total size of data generated in the past two years [1]. According to Soprasteria, information technology consulting company, they expect that the data size generated in 2020 will be 44 times greater than in 2009, resulting in 44 zettabytes in 2020 [2]. Therefore, the companies in big data industry, such as Oracle, IBM, and Google, will try to develop analysis software to handle large amount data. Especially, the company tries to make statistical software such as BigR [3, 4], Rhadoop [5] and Spark [6], which analyze the massive amount of data in a fast, convenient and efficient [7]. This software is based on Hadoop system [8] to handling rage amount of data.

According to the Wikibon [102], the value of big data market accounts for approximately 92.2 billons. Wikibon divided the big data market into three parts: big data service (40% of all big data market's revenues in 2015), hardware (31%) and software (29 percent). Among the

three, the big data software is projected to grow faster than others. The ratio of big data software will be 46% of the big data market in 2026.

The big data comes from various sources such as social media, stock exchange, transportation, search engine, healthcare, etc. Data sources are categorized into three types: 1) structured data such as relational data, 2). semi-structured data such as XML data, CSV data, and 3) unstructured data such as Word, PDF, text, media, etc. The unstructured data is difficult to handle using traditional methods, yet the amount of unstructured or semi-structured data are increasing than the amount of structured data. Hadoop is well suited in handling such data, and therefore, it has received a huge attention by data scientists. Apache Hadoop (hereafter, Hadoop) is a software framework in big data analytics, and performance of this framework has a substantial impact on the entire system [9]. It has been widely used by scientists and industry.

1.2 Hadoop Performance Research

Recently, people have started focusing on high speed performance of big data analysis to apply a result of the analysis into the real world. Company and researcher have shifted their focus towards the real-time strategy with big data to get more values and benefits. They have studied to increase Hadoop performance to analyze data of various aspect and huge size.

There are three major ways to improve Hadoop performance: 3rd party software on YARN, data locality in Hadoop and Hardware. Figure 1-1 shows the performance research area in Hadoop system. The words with blue color in Figure 1-1 represents the 3rd part software research like Spark [6], MapR [103] and so on. By the 3rd part software such as in-memory system, big data scientists can analyze big data in real-time. However, there is some limitation

such as cost and size of data. Bascially, memory is more expensive than a hard disk to store a data. Therefore, analysis data is limited by the size of memory.

The green color in Figure 1-1 represents the hardware research such as Solid-State Disk [104], InfiniBand [105] and so on. The basic concept of Hadoop is that it uses cheap machines to handle such a big data. Whenever the data size is enlarged, the hardware's cost will be dramatically increased to support networks, hard disk, memory and so on. So, Hadoop pursues to use cheap machines, 4 cores, 8 Gbyte memory and 2 hard disks, for installing Hadoop clustering.

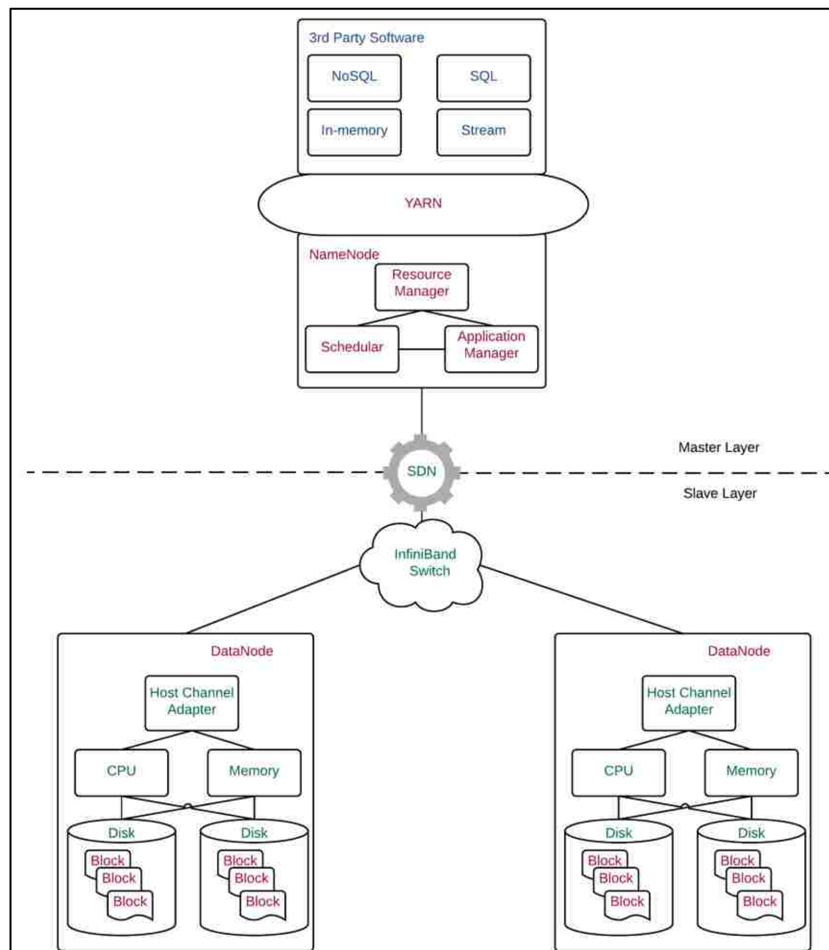


Figure 1-1. Composition of Hadoop system with physical hardware and 3rd party software

The Hardware and 3rd party software required extra costs or didn't give impact on the core of Hadoop system.

One important principle of Hadoop is "Moving Computation is cheaper than Moving Data" [1], as shown in Figure 1-2. A requested computation by an application is much more efficient if it is executed near the data on which it operates. This is especially true when the size of the data set is large [1]. This minimizes network congestion among nodes in MapReduce and increases overall throughput of the system. The assumption is that it is often better to migrate the computation closer to where the data is located rather than moving the data to where the application is running like in Figure 1-2. Hadoop Distributed File System (HDFS) provides interfaces for applications to move closer to where the data are located [10]. Researchers have

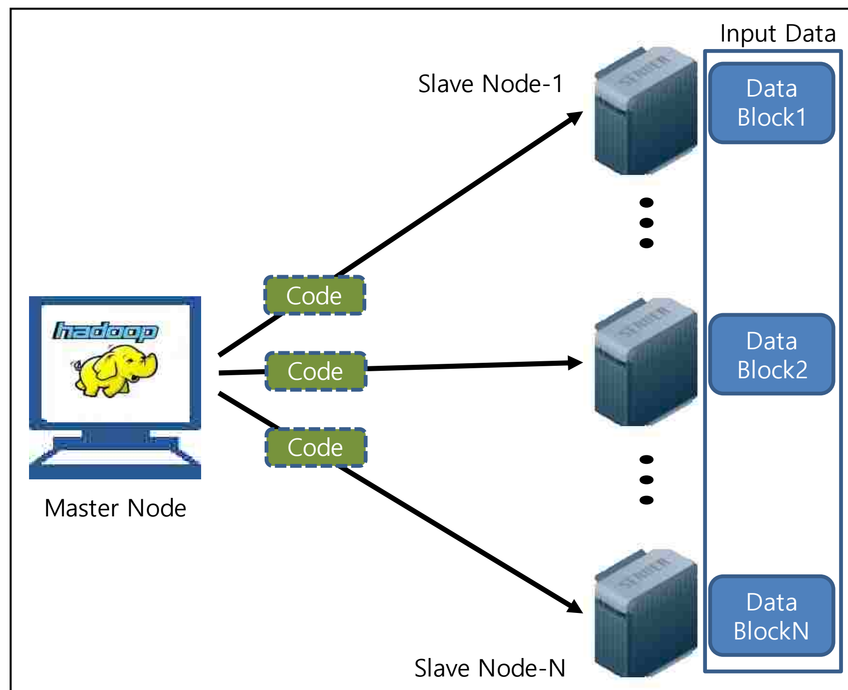


Figure 1-2. Moving code to Slave nodes.

been trying the minimize network congestion, increase transmission rates, to increase the overall throughput of the Hadoop system.

1.3 Overview of Our Works

Our research explores the concept of data locality. The red color in Figure 1-2 shows the data locality research on Hadoop system. The data locality research can improve Hadoop performance without extra cost by addressing inside of Hadoop system such as YARN, Hadoop Distributed File System (HDFS), MapReduce. (The detail information of Hadoop will be expalined in Chapter 2.). Therefore, We have been studying the data locality to give broad impact in big data software without any extra cost.

There are two data locality concepts in Hadoop. One is the initial block locations on HDFS. The information of initial blocks location is used for allocating task into mappers and reducers. The other is the partitions. After finding the task on mapper, partitions are sent to reduers based on the key. The process of moving partitions creates a network congestion between mappers and reducers. Traditional data locality research [12-22, 27,31,109-111] studied the initial location of blocks to allocate task into mapper. The research tried to increase the data locality between mappers and data nodes. In a research [111], the performance of Hadoop was improved by up to 28%, when tested with 10 GB of input data on WordCound and Terasort Benchmark [50].

However, there are many limitations on the data locality research. (Chapter 4.6.) Most of the data locality research focused on early stages (i.e., Map) and overlooked later stages (i.e., shuffle) in MapReduce, even though the locality can be increased in later stages. Most researches in the data locality addressed only the initial locations of blocks on HDFS. They generally use a

strategy using a fixed number of replicas on HDFS, same block for the replicases on HDFS and default block placement polish on HDFS. This is because they start optimizing the performance after uploading the input data into HDFS.

In this research, we applied the data locality to all stages of MapReduce. We developed a new concept of deep data locality (DDL). Furthermore, we have explored two types of DDL, that is, Block-based DDL (Chapter 6.2) and Key-based DDL (Chapter 6.3). The DDL concept can be applied to multiple stages of MapReduce regardless of application types, number of containers, or node status. DDL concept can be combined with other data locality methods, such as 3rd party software or speciaized hardware.

Another contribution of our work is the computational models and simulation tools that can predict Hadoop performance by considering various factors. Predicting hadoopo performance has been difficult due to several reasons. First, it is hard to predict node's status in each stage. Hadoop uses multiple nodes to process big data. Also, each node executes several different daemons and the. status of nodes changes constantly. Consequently, it is hard to predict each node's status and daemons. Second, each stage in MapReduce uses a different number of containers, such as mapper and reducer, to process the data dependindg on the configuration. The nodes status and the locality on Hadoop can be changed based on the configuration and locations of blocks. Lastly, the blocks in HDFS are used differently depending on the types of applications. And the locality of blocks in HDFS are changed accordingly. We created a computational model that considers these factors, and developed simulation software that integrates the model.

CHAPTER 2. HADOOP SYSTEM

Chapter 2 introduces the background of Hadoop system. Understanding the background of Hadoop is important to know the reason of the data locality caused in the Hadoop system. There are four modules in Hadoop system such as Hadoop common, YARN, MapReduce, and HDFS. Among the four modules, the HDFS and MapReduce are illustrated in this chapter. HDFS and MapReduce are crucial elements for understanding the data locality. All blocks are in HDFS and the data locality is caused when MapReduce process the blocks in HDFS. First, this chapter explained the Master-Slave architecture to understand the physical Hadoop structure. Second, the detail of HDFS is explained about the block system in HDFS and block replication. Understanding HDFS helps to explain the data locality in chapter 3 and 4, and Block-based DDL in chapter 6. Lastly, the process procedure of MapReduce is illustrated by dividing the MapReduce into eight stages. The process procedure of MapReduce is related to the Hadoop performance analysis model in chapter 5. The map, transfer, and reducer among the eight stages must be understood for comprehend the Hadoop performance analysis model in chapter 5.

Hadoop is an open-source software framework that works as distributed processing of big data [10]. Hadoop offers file system and operating system abstraction written in Java. Big data includes various data types – such as unstructured, semi-structured, and structured data – characterized by their size. Analysis of the unstructured and semi-structured data sets are complex process to be accomplished using a single desktop computer. Utilization of the Hadoop system is essential to solve such problems. Hadoop is mainly consisted of four modules: Hadoop

Common, Hadoop HDFS, Hadoop Yet Another Resource Negotiator (YARN), and Hadoop MapReduce.

Hadoop Common is a basic module for supporting the system. It refers to common utilities and essential Java libraries needed by Hadoop. Hadoop Common includes a base core of the framework that supports services and processes on Hadoop, such as abstraction of the basic operating system and associated file system.

Hadoop YARN module is a framework for job scheduling and application management. YARN was released in the upgraded Hadoop version 2.0. In YARN, Resource Manager (RM) supports scheduling, and Node Manager (NM) are used for monitoring in the Hadoop system. RM receives the job from the client, schedules the job, and allocates it to slave nodes based on the slave's status. Each slave node has NM, a framework for monitoring resources such as CPU, memory, disk, network, and so on. NM responds to the request of RM about status. By this function, Hadoop achieve scalability, multi-tenancy, cluster utilization, and compatibility.

HDFS is a file system for Hadoop. Input data is split into smaller blocks and distributed in a Hadoop cluster. HDFS use three VM daemons – such as NameNode, Secondary NameNode and DataNode – to manage the small pieces of block in Hadoop. MapReduce can be executed on the blocks using mapper and reducer, and this provides the scalability that is needed for big data processing [11].

2.1 Master-Slave Architecture

Hadoop has master-slave architecture. Master layer manages the application and slave-node resources. Slave layer stores the data and processes the application with the data. The master node is normally called NameNode and it works mainly for the user and MapReduce,

because, the node stores all HDFS metadata that contains information on block locations, namespace, identification, and replicas about blocks in slave nodes. In Hadoop version 1.0, JobTracker in NameNode manages all slave nodes and applications to process the job. JobTracker distributes the job into slave node through Task Tracker in version 1.0. JobTracker schedules and manage the entire job. JobTracker calculates a number for Mapper and Reducer to execute the job. And, JobTracker selects TaskTracker in Slave Node. Most of the data locality scheduling research in Map stage started with JobTracker and TaskTracker to increase the locality in Map stage [12-22].

Each TaskTracker in slave node manages mappers and reducers to process the

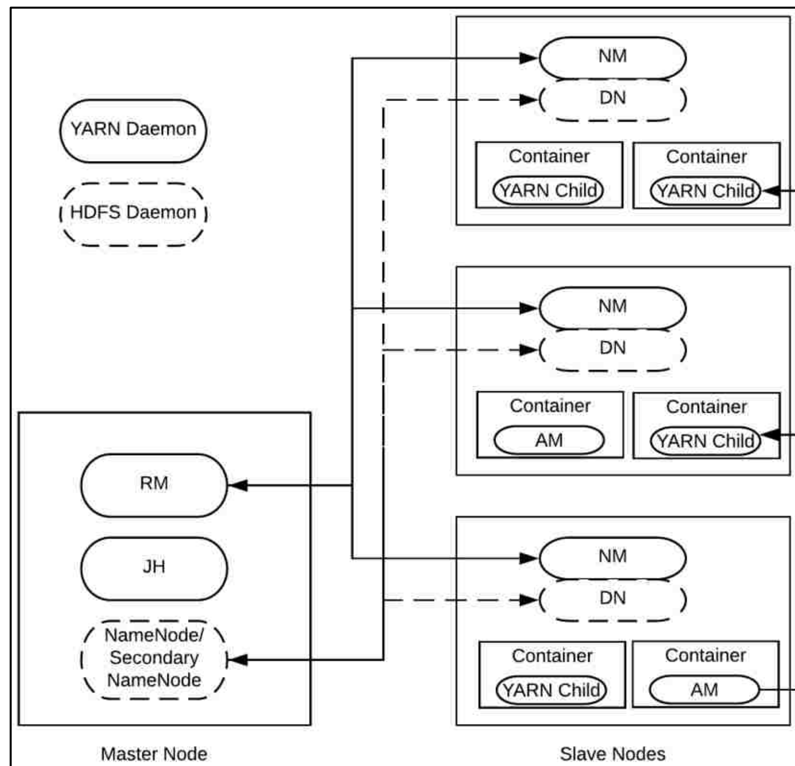


Figure 2-1. Hadoop VM Daemon.

MapReduce job. Hadoop version 1.0 has a scaling issue and is limited to run other frameworks by JobTracker. In the updated Hadoop version 2.0, JobTracker's role is divided into mainly three different type of daemons to manage clusters and applications by YARN. These include RM in master layer, the Job History (JH) server, and NM in slave layer. TaskTracker has been replaced with Node Managers (NMs) and Application Masters (AMs). Here We skip the JH server because it doesn't improve the performance of Hadoop in YARN.

YARN mainly has three different types of daemons: RM, NM, and an application manager (AM) (Figure 2-1). RM in master node is the master that manages all NMs and schedules the applications. RM receives the report from NMs through the heartbeat of each NM. Based on this information, RM launches the AMs in slave nodes and scheduling to process allocated applications. Task Tracker's role in version 1.0 is split into NM and AM. Table 2-1 shows how to change the name and role in the Hadoop daemon. Each slave node has its own NM to take instructions from RM, manage resources, and report status to RM. AMs work with NMs to launch containers. There is one AM per application. Each container takes the mapper and reducer role using the YARN child daemon. Current data locality research with scheduler uses

TABLE 2-1. Hadoop Version 1 vs. Hadoop Version 2

	Hadoop version 1	**Hadoop version 2**
Cluster Manage	JobTracker in Master Node	Resource Manager in Resource Node
Application Manager	JobTracker in Master Node	Application Master in Slave Node
Task Manager	Task Tracker in Slave Node	Application Master in Slave Node
Task	Mapper/Reducer in Slave Node	Container (YARN child) in Slave Node

YARN to allocate tasks into containers that have the data stored locally on disk.

2.2 Hadoop Distributed File System

Hadoop uses HDFS modules for a distributed file system. HDFS is the file management layer of Hadoop. It is a Java-based distributed file system for big data. HDFS is more highly fault tolerant and requires low-cost hardware than other distributed file systems. By HDFS, Hadoop can access input data with high throughput and handle large data sets. The HDFS cluster has a single NameNode that manages the file system namespace and regulates access to blocks as a master node in Hadoop version 1.0. Some of the roles are moved to the resource manager in version 2.0 to manage the slave node during the process for scaling the Hadoop system. HDFS mainly has three daemons to manage the blocks – such as NameNode, Secondary NameNode, and DataNode (DN). The NameNode is a centralized system to handle DNs using metadata which contains all the information of related block such as locations, name, identification, and so on.

Secondary NameNode is the backup for NameNode with fsimage. When NameNode restarts or stops, the entire Hadoop system in version 1.0 does not work at all. Secondary NameNode serves as a master node to overcome this issue. The secondary node puts a checkpoint in the filesystem, helping NameNode to function better or recover. Figure 2-1 shows the HDFS daemon in master node and slave node. Each slave node has one DN to manage the data set and report the block information to NameNode periodically. DNs are responsible for serving read and write requests from the file system's clients. The DNs also perform block creation, deletion, and replication upon instruction from NameNode.

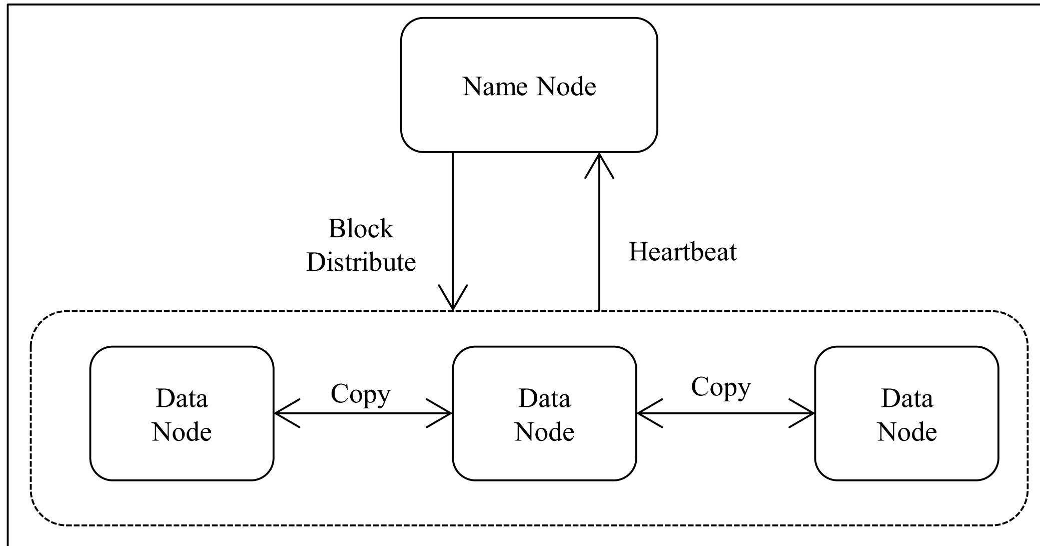


Figure 2-2. Basic architecture of HDFS.

Figure 2-2 shows the basic architecture of HDFS which stores the file as block units. HDFS splits large files into blocks (64MB or more) and distributes the blocks to data nodes. For example, if the file size is 1 gigabyte, the file divides by 128MB and each block stores at slave nodes. The block size in HDFS is set at “hdfs-site.xml” configure file by “dfs.blocksize.” The blocks normally are duplicated to other slave nodes. The number of duplicating is set at “hdfs-site.xml” configure file by “file.replication.” The detail configuration of HDFS is in the appendix. The duplicating location is selected by block placement policy in HDFS. The default block placement policy in HDFS is the second duplicating block and has to be stored in a different rack of the server. Accordingly, HDFS manages files as block units and copies them to several slave nodes. Block information – such as size, location, and replication – is stored at master nodes using a metadata file.

This block system in HDFS and replication are important to data location research which seeks to improve data locality in Map stage and Shuffle stage.

2.2.1 Block System in HDFS

One large file is difficult to read using a single machine [9], but small files are easy to read on a single machine. Also, HDFS supports write-once, read-many semantics on files. Hadoop splits the file into several components and sends the code to process the application like Figure 2-3. HDFS stores the original data to be divided into several blocks which will be distributed into several DNs. The block size can be set by the HDFS configuration file which contains basic setup for HDFS (hdfs-site.xml). The default block size is 128 MB in HDFS configuration in Hadoop version 2.0. Each DN manages the block and master node stores the block information using metadata.

For example, if the block size is set to 128 MB in the HDFS configuration and the original data size is 3 GB, then HDFS splits the input data into 24 blocks of 128 MB each. Additionally, HDFS makes replicas of the blocks (detail in next section). Each block is replicated

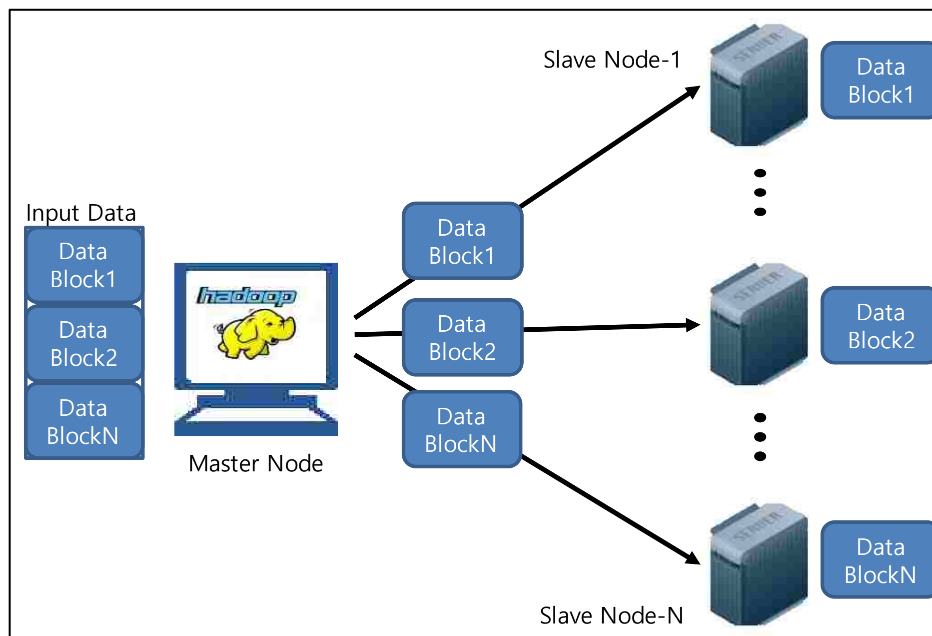


Figure 2-3 Hadoop split input data into several blocks.

a number of times based on the HDFS configuration to ensure high data availability, fault tolerance, and the ability to run on commodity hardware. When RM has requested the application with 3 gigabytes from a user, RM normally selects a container which has a block to process the application. Each container processes less data than the original large data file.

2.2.2 Block Replication

HDFS reliably stores the large data file across nodes in clusters. Each component of input data is stored as a sequence of blocks. All the blocks are the same size except the last block. Each slave node makes replicas of the block for fault tolerance. The number of replicas is set at

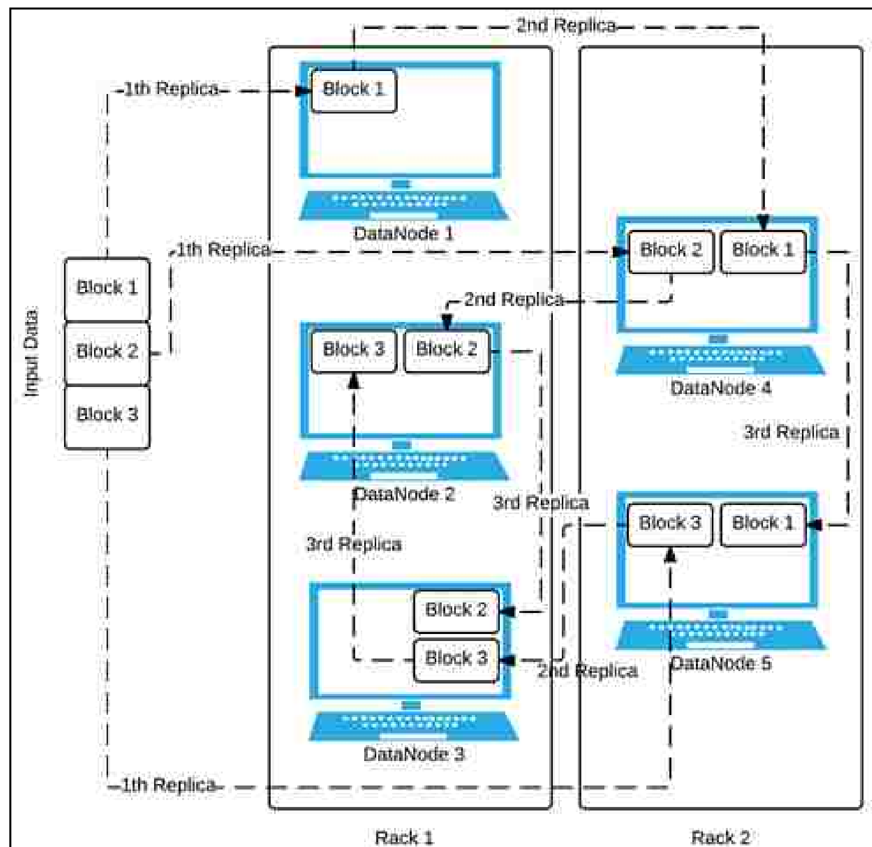


Figure 2-4. Basic replica placement policy on HDFS.

replica factor in hdfs-site.xml (see Appendix). The replicas are stored at different nodes. The replicas follow the HDFS's placement policy – such as rack awareness to distribute the replicas for data reliability, availability, and network bandwidth utilization [23]. Figure 2-4 shows the replica's role in Hadoop.

For example, if there are three replicas of a block, the first replica is stored at the rack 1 slave node and the first replica makes a copy in the rack 2 slave node. The replica in the rack 2 slave node duplicates itself at a different node in same rack (rack 2). Hadoop provides reliable storage without backup data by the replicas. Hadoop improves data locality to process big data via replicas in HDFS.

The purpose of a rack-aware replica placement policy is to improve data reliability, availability, and network bandwidth utilization. For the most part, the blocks spread out to nodes. Communication between two nodes in different racks has to pass through switches. In most cases, network bandwidth among machines in the same rack is greater than network bandwidth among nodes in different racks. Some data locality research uses the replica placement policy because Hadoop can keep fault tolerant and improve data locality by using speculative task [24, 25] with the replicas [26]. When one of the nodes that processes the job with mapper or reducer is down, the other node can replace the node's job as a speculative task with the replica. If there is no replica in another node, the application fails to finish the job.

2.3 Basic Architecture of MapReduce

MapReduce is a programming model and an associated implementation for processing in parallel on the HDFS [9].

Figure 2-5 shows the basic architecture of MapReduce in Hadoop version 1.0. Job Tracker in master node manages job scheduling. Job Tracker distributes the job to the selected TaskTracker in slave node. Task Tracker performs the job using Mapper and Reducer. The job, an actual program working for a user's request, is divided into several slaves by JobTracker. JobTracker is located on master node to schedule jobs and track tasks. The distributed job is

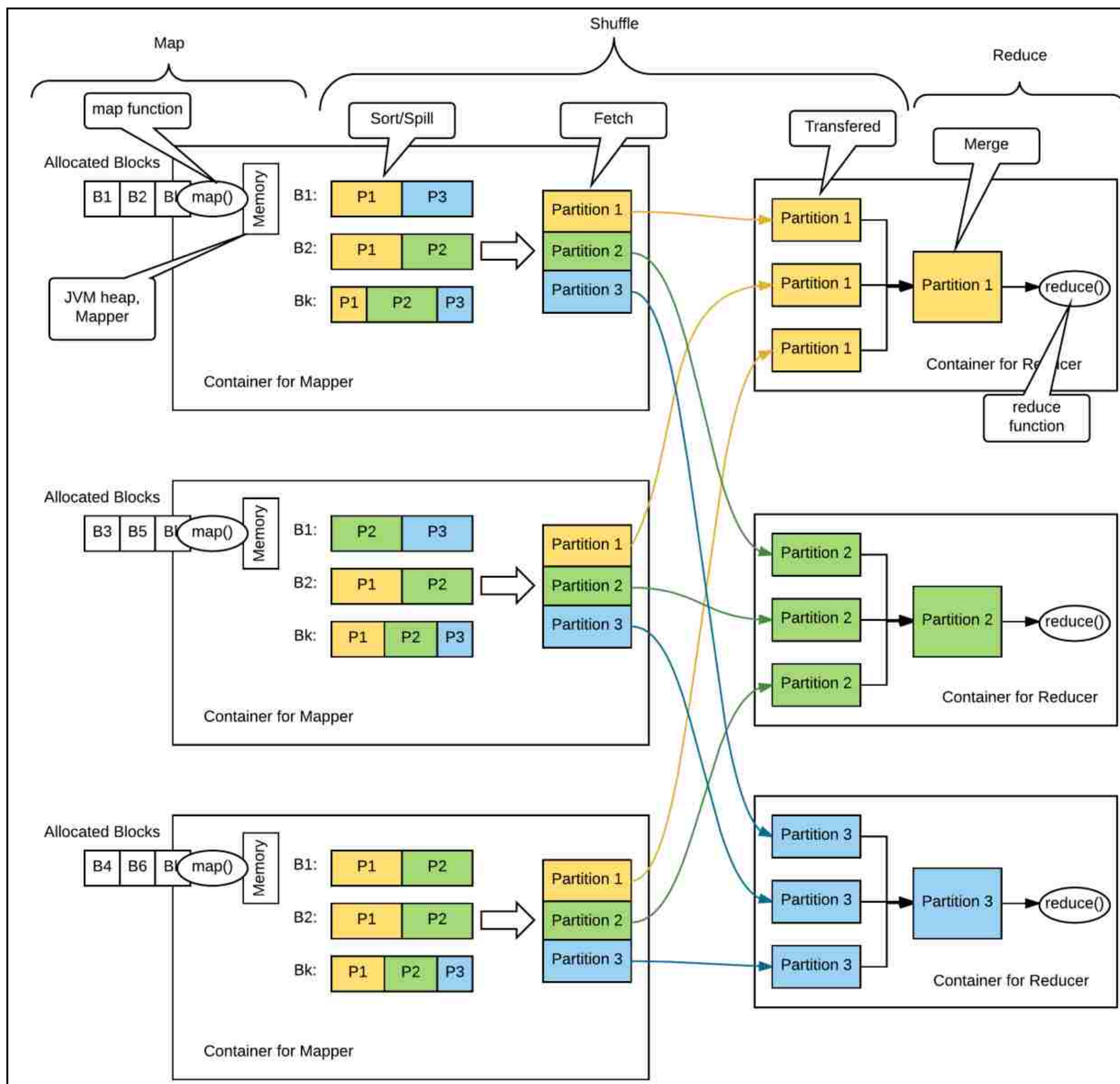


Figure 2-5. Basic architecture of MapReduce in Hadoop version 1.X.

called a task, which is the execution of a mapper or a reducer on a block. Mapper is a middle process to map the input key-value pairs to a set of intermediate key-value pairs. Normally, the task allocated to mapper depends on data locality in map stage. Each slave node has TaskTracker to report the status of mapper or reducer to JobTracker. After map stage, mapper sorts the intermediate output by key to send it to Reducer. This phase is called the sort/spill phase.

When one of the mappers finishes the sort/spill phase, the mapper sends the intermediate output to reducer for shuffle stage. The Shuffle phase works on the network. Figure 2-6 illustrates shuffling the pair by key and value. By shuffling, each reducer gathers a specific key and value without repetition of the key.

When the intermediate output reaches slave nodes, each slave merges the intermediate output by key. This is called the Merge phase. When the some amount of merge files is stacked the reducer starts to process the batch job. The reducer stage processes the intermediate output. After processing, reducer produces a new set of final output. This final output is stored in the

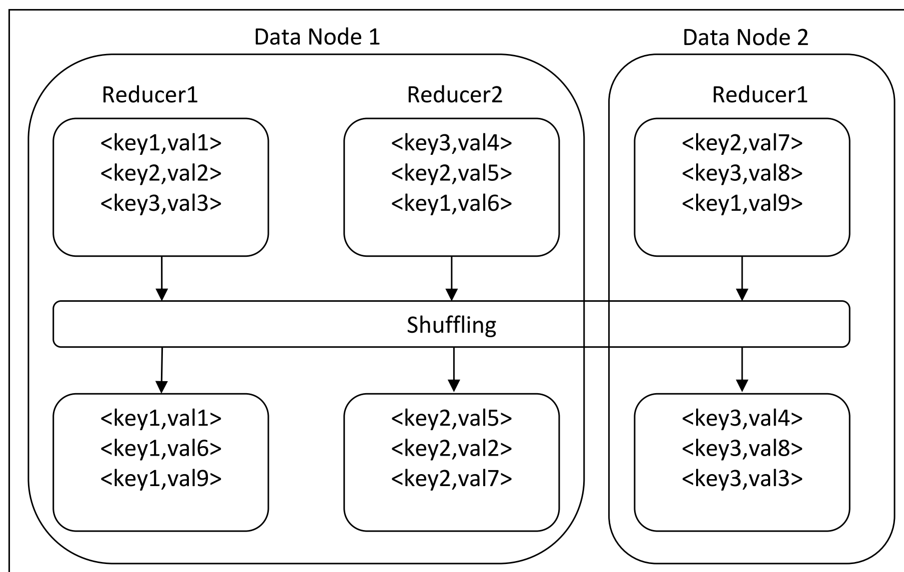


Figure 2-6. Hadoop Shuffling.

HDFS as a result for the client's request.

The MapReduce stage is important to improving the processing time of MapReduce using data locality. In next section, We detail use of MapReduce with Hadoop version 2.0 to understand data locality research in Hadoop.

2.3.1 MapReduce on Hadoop Version 2.0

This section presents an overview of current research on Hadoop performance enhancements and the background on MapReduce and YARN. We will briefly describe the workflow of MapReduce as illustrated in Figure 2-7 and survey the related research works.

YARN is divided into two major parts, one for job scheduling using Scheduler, and the other for resource management via Application Manager. MapReduce is divided into three major stages. The first stage is Map which reads input data and emits key/value pairs in the data node. The second stage is Shuffle which redistributes data to reducer based on the output of map. Sort/Spill and Merge stages are typically included in the Shuffle stage. The third stage is Reduce

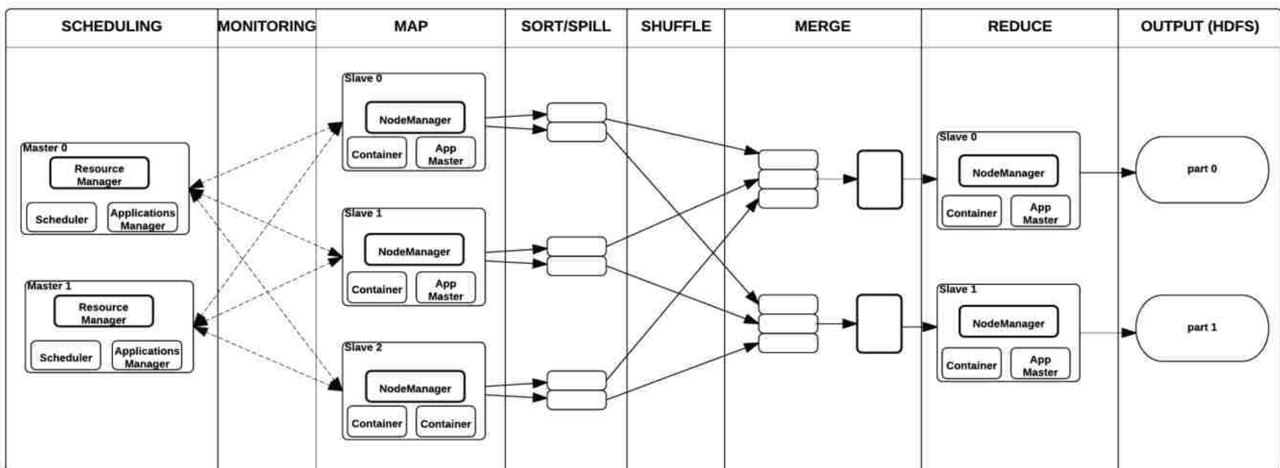


Fig 2-7. The workflow of MapReduce on Hadoop Version 2.0.

which accepts a key and all the values having the same key and emits the final output to the data nodes. The workflow of MapReduce is further divided into the following eight stages.

1) Scheduling: When a client submits a job to the Master node, the Resource Manager in the Master node executes the Application Manager which decides whether to accept the job or not. Application Manager in the master nodes manages the Application Master (App Master in Figure 2-7) which handles the containers in the job. When a master node receives a client's request, the Application Manager selects one Application Master among the slave nodes to execute the job. The Resource Managers schedule the job using scheduler if the job is accepted. The scheduler creates several Containers in the data nodes based on the Resource Tracker which stores the configure information and the status of resources in the data node, to divide the job into several tasks. Many previous researches have studied the scheduler in order to optimize the usage of resources in the data node to the highest level using container allocation [27], data locality [28, 29], storage [30], optimizing configure of Hadoop [31] and workload balancing [32].

2) Monitoring: Each node has a single Node Manager which reports the status of the node, such as CPU, memory, and disk status to the Resource Manager using Heartbeat, the communication method among containers, Application Master and Node Manager. Scheduler makes the job scheduling based on the information of Heartbeat [33] and the Heartbeat has an influence on the initialization and the termination of a job [34]. Application Master reports a hardware and software failure to Resource Manager via Heartbeat when Application Master detects the failures from containers and data nodes [35]. Therefore, the Monitoring stage is important to assist scheduler and to make the system highly fault-tolerant.

3) Map: Mappers in containers execute the task using the data block in data nodes. This is a part of the actual data manipulation for the job requested by the client. All mappers in the activated containers by Application Master execute the tasks in parallel. The performance of mapper depends on scheduling [27, 31], data locality [28, 29], programmer skills, container's resources, data size and data complexity.

4) Sort/Spill: The output pair which is emitted by the mapper is called partition. The partition is stored and sorted in the key/value buffer in the memory in order to process the batch job. The size of the buffer is configured by Resource Tracker and when its limited is reached, Spill is started. This stage has been studied as part of the Shuffle stage because the performance of this stage depends on the programming in map and hardware resources. However, it may be worth to study it separately.

5) Shuffle: The key/value pairs in the spilled partition are sent to reducer in Slave nodes based on the key via the network in the Shuffle stage. Most of the network problems occur in the Shuffle stage due to the huge volume of data. Increasing the network performance is a big issue and researchers have approached it from Software Define Network (SDN) [36], Remote Direct Memory Access (RDMA) [37], and Hadoop configurations [38], etc.

6) Merge: The partitions in the partition set is merged at container which works as a reducer in order to finish the job. This stage has been usually studied along with the Shuffle stage, such as in-memory with compression [39, 40].

7) Reduce: The reducer in the Slave nodes processes the merged partition set to make a result of the application for the client's request. All reducers in the containers activated by Application Master run the tasks in parallel. The performance of reducer depends on

scheduling [27, 31], data locality [28, 29], programmer skills, container resources, data size, and data complexity, as was the case in the Map stage. However, unlike in the Map stage, the Reduce stage can be improved by in-memory computing [39, 40, 41, 42].

8) Output: The output of reducer is stored at the slave nodes. This output data may be still big even after going through Map and Reduce stages. Therefore, data compression is needed in the Output stage. Various methods of compression have been studied to reduce the data size in order to transfer the data among cluster nodes [32, 41]. Although this stage has not drawn much attention due to the maturity of compression algorithms, it is still important to improve inter-node communication efficiency.

We note that there are other areas affecting the performance, not in the workflow of MapReduce, such as storage method of big data [30], garbage collection [40], in-memory computing [39, 40, 41], pre-partitioning [44], etc. Most of the research related to the performance of MapReduce affect only one specific stage out of all. For example, the SDN [36] on the Hadoop increases the performance of shuffle stage. The research of Hadoop scheduling [27] and data locality [28, 29] affect Map stage to increase the speed of Hadoop. Unlikely, the LNBPP extends Map stage to Shuffle stage in the workflow of MapReduce to improve the performance of MapReduce.

CHAPTER 3. DATA LOCALITY

Chapter 3 explain the data locality. The overall concept of the data locality research will be addressed in the chapter. There are three data localities on Hadoop, such as data-local, rack-local, and off-rack. This chapter clarified the detail of the three data localities on HDFS. This chapter showed how the data localities appear on the HDFS and the effect of the rack-local on the Map stage. The data localities cause the network congestion during the MapReduce processing. Understanding the data localities is important to comprehend the traditional data locality research and the importance of the data locality research on Hadoop. At the end of the chapter, the basic scheme of data locality is explained to easily understand the traditional data locality in chapter 4.

Achieving data locality at one of the stage in MapReduce – map, shuffle and reduce stage – is quite intuitive. The input data is split into small data blocks and distributed to slave nodes. We refer this type of data as locality SDL (Shallow Data Locality). However, the data gets relocated at Shuffle stage and the Reduce stage cannot utilize any locality. If the data is pre-distributed to the appropriate Reduce nodes, it will reduce the Shuffle time and improve the performance. We refer this type of data locality DDL (Deep Data Locality). However, due to the high complexity of Hadoop configuration and the difficulty of cross-optimization between Hadoop and MapReduce, the data locality on shuffle stage has not been studied widely yet. To utilize DDL fully, an integrated approach between Hadoop and MR is needed.

Researchers have tried to modify the unbalanced block placement in slave nodes to make the performance more consistent across the slave nodes using data location [28, 31], scheduling

job [31] and so on. They focused on the resource of slave nodes such as CPU and memory to increase the utilization of the resources in the slave nodes to the highest level. Although those efforts improved the performance to some degree, the performance has not been fully optimized because they have not considered the locations of intermediate outputs in Shuffle and Reduce stages. Before addressing the main idea of DDL, We introduce the data locality in this chapter to understand DDL.

The data locality impacts performance of Hadoop such as speed, reliability, analysis, availability, network bandwidth utilization, especially distributing data and data management components. Data Locality allows a Map program to be executed on the same node where the data is located to reduce network traffic. During this process, to increase the data locality, Hadoop makes several replicas of the data blocks, and distribute them to multiple slave nodes. This duplication increases not only the data availability but also data reliability in case of data loss.

3.1 Basic Concept of Data Locality

Before processing big data in Hadoop, the user must upload the input data into HDFS. The input data is divided into several chunks that are stored in slave nodes and made into several replicas in different nodes [10]. Therefore, processing data in Hadoop works on HDFS using blocks in slave nodes. The type of block location can be divided by three based on the location of the blocks. Fig 3-1 shows the three types of locality in HDFS: data-local, rack-local, and off-rack.

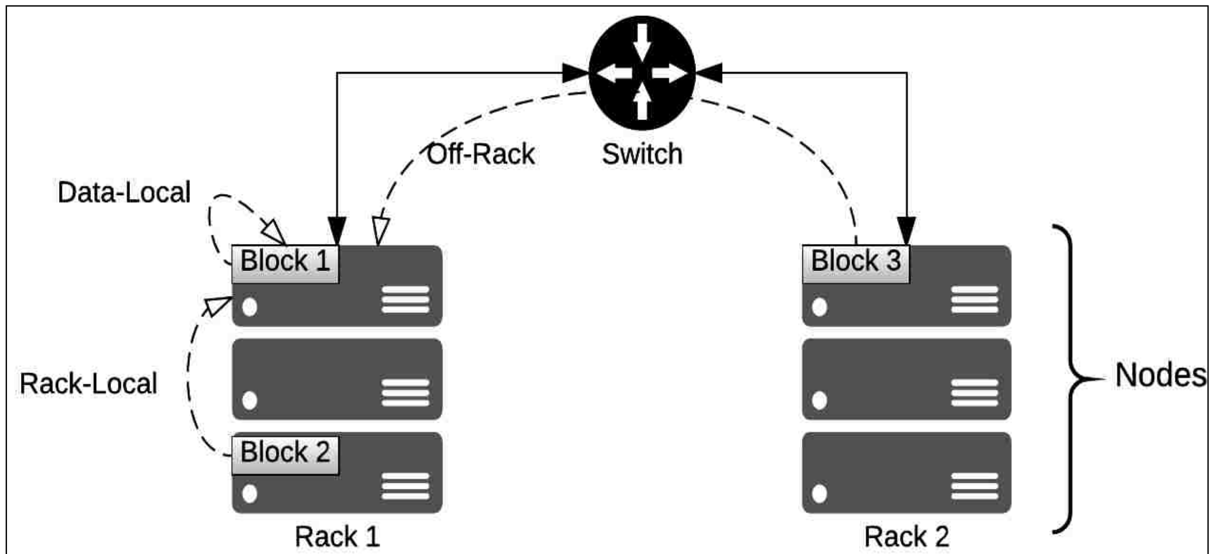


Figure 3-1 Data locality in HDFS.

1) *Data-local (local disk)*: The data block is on the local disk. The node can process the data block without a copy of data from other nodes as in Figure 3-1's Block 1 in slave node. So, there is no overhead of network traffic. Mostly, YARN tries to allocate a job to the node which stores the data block in data local. Most data locality research has tried to keep data-local in MapReduce.

2) *Rack-local (in-rack disk)*: The data block is on another node in the same rack. So, the node has to copy the data block from another node in the same rack as in Figure 3-1's Block 2. When YARN does not receive any heartbeat from the node that has the data block for processing the job, YARN allocates the job into another node which is idle even though the node does not have the data block. The rack-local directs network traffic among nodes to copy source block. In the shuffle stage, there is the rack-local in shuffle stage because of partitions in mapper. Mapper has several partitions based on key, and the partitions have to move to reducer based on key, eventually one of partition has to

move other nodes. The scheduler research in Hadoop tries to allocate the job to a node in the same rack or local-disk to reduce network traffic.

3) *Off-rack (off-rack disk, off-switch, rack off)*: The data block is not on the same rack but on another rack. Sometimes, the processing node has to copy the source block from another node in another rack like Figure 3-1 's Block 3. If the node does not have Block 3 to process a job in MapReduce, the node has to copy Block 3 from another node that has Block 3 to process the job. The off-rack creates the largest delay in network traffic among data locality in Hadoop. There are some off-rack situation in Hadoop on the cloud system because the cloud system supports resource using VM with several server. So, the literature [45-49] tries to gather the slave nodes in the same server, if the nodes have the sequence of input data.

Data locality mostly is applied in map, shuffle, and reduce stages in big data processing, especially in MapReduce. YARN, for example, selects the slave node which has the blocks like data-local to process a job in map stage. By sending the code to the slave nodes, Hadoop can improve locality in map stage. YARN has considered locality when YARN selects a container as a mapper.

3.2 Example of Data Locality with Test

To understand the movement of blocks during shuffling, We have tracked the locations of blocks and containers on HDFS. We uploaded 2.2GB of customer review data from TripAdvisor on HDFS composed of 20 slave nodes. The data is divided into 18 blocks and 6 replicas are created for each block, resulting in 108 blocks. They are spread to 20 slave nodes by HDFS.

Table 3-1. Block location in slave nodes

Block \ Node	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**Total**
0		●				●									●	●		●	●		6
1					●		●							●				●	●	●	6
2	●		●		●						●			●		●					6
3		●	●						●		●			●				●			6
4				●		●		●					●		●					●	6
5		●					●			●	●			●					●		6
6		●									●	●		●				●	●		6
7	●	●					●			●			●			●					6
8		●				●						●		●		●			●		6
9			●				●			●	●				●	●					6
10					●		●	●	●		●									●	6
11		●	●			●		●				●					●				6
12						●	●	●									●	●	●		6
13								●	●	●				●				●		●	6
14			●	●	●							●					●	●			6
15			●				●	●			●					●				●	6
16		●		●		●											●	●		●	6
17								●	●		●		●		●			●			6
Total	2	8	6	3	4	6	7	7	4	4	8	4	3	7	4	6	4	9	6	6	108

Table 3-1 shows the locations of those blocks in 20 slave nodes on HDFS in one test run. This block assignment is different in every execution. According to Table 3-1, the blocks are randomly spread by the DBPP on the HDFS. Therefore, each slave node has a different number of blocks ranging from 2 to 9. We tracked the blocks when MapReduce processes the job which is submitted by clients. After executing the job 20 times, We observed that the performance of MapReduce fluctuates significantly and We found a correlation between the number of RLMs and the performance. RLM (Rack-Local Map) is the number of map tasks that doesn't have a block on its own node but has it in another node on the same rack. The RLM introduces an overhead while executing the Map stage because the block has to be copied from another node to

Table 3-2. Effect of RLM on T_M and T_P on DBPP

Number of RLMs	TM (sec)	Tp (sec)	# of test runs
2	27.974	51	3
3	29.274	53	3
4	36.620	62	3
5	37.828	64	3
6	38.105	65	5
7	41.461	71	3

**** The values are the average of the test runs.***

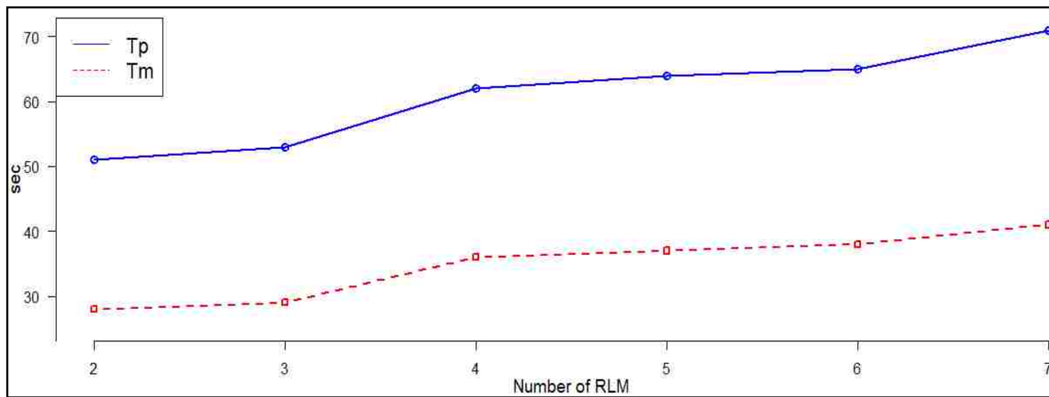


Figure 3-2. Effect of RLM on T_M and T_P on DBPP.

the node running the task. Because different blocks are assigned for each node in every new execution, each execution generates different number of RLM. Table 3-2 and Figure 3-2 show the result of the test runs where TM and TP are increased as the number of RLM is increased. Table 3-3 shows one example of activated slave nodes which have activated container. In some cases, the nodes do not have the necessary blocks. For example, according to Table 3-1, node 17 does not have block 9, so it needs to copy the block that is called RLM. There are three RLMs in this case. Likewise, blocks 12, and 14 are copied by slave nodes 13 and 13 respectively. The number of RLMs varies dynamically for each execution of a job, which is determined by Application Master.

Table 3-3. Allocated slave nodes to process MapReduce on HDFS

Block	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Assigned Node	1	17	13	13	12	13	13	12	13	17	17	1	13	17	13	17	17	12
Has the block?	Y	Y	Y	Y	Y	Y	Y	Y	Y	**N**	Y	Y	**N**	Y	**N**	Y	Y	Y

3.3 Basic Scheme for Data Locality

According to related work, most data locality research has focused on scheduling to reduce the data transferring among nodes because of version and previous distributed system research. Therefore, to understand basice scheme of data locality is very useful understanding the data locality. In this section, We show the basic scheme of data locality scheduling in Hadoop. The fundamental principle of locality scheduling is reducing the amount of data transferring among nodes when processing moves to the next stage. The scheduler selects the node that has a data block to process data. Algorithm 3-1 shows the principle to achieve the locality in MapReduce. This scheme does not consider the resource of the node to select the mapper and reducer, but it is sufficient to show a basic scheme of locality schedule for understanding. The first while loop in Algorithm 3-1 keeps rotating until all the block is allocated to the slave nodes. The second while loop finds the data-local node to allocate the block in mapper. The data-local nodes, that have real data in local disk, are selected to increase data-local. If the block is in the node, the node is selected as a mapper to process the block. If the block is not in the list of nodes, one of the list nodes will be allocated based on the node that has the block in the same rack. If the node cannot be selected in the local disk, the scheduler selects the node from the near node which has real data to reduce the off-rack. After finishing the map stage, there are several

partitions in each node. To increase the data-local in the for loop, the node, which has most of the key, is selected as a reducer.

The next chapter explained how the performance of Hadoop is improved by traditional data locality. The basic scheme of data locality will be useful to understanding the data locality scheduling.

Algorithm 3-1: Basic scheme for scheduling.

```
     Input: D (the list of block number in the data)
     Output: P (Partition), O (Output in Reducer)
     // b: list of block numbers, L: list of nodes and b, n: node
     //m: mapper, r: reducer, M: mapper list, R: reducer list, k: key
1    Initialization: P ← 0, R ← 0, b ← 0, N ← 0, O ← (0, 0), M ← 0
     // select node for mapper
2    while D < > 0 do
3      | b ← get_number_blocks (D)
4      | n←get_next_node(L)
5      |  while b <> 0 || n <> 0 do
6      |  | if b in n
7      |  |   | Add_mapper(M,n,b)
8      |  |   | b = 0;
9      |  | end
10     |  | n←get_next_node(L)
11     |  end
12     | If (n ==0 &&b <>0) Add_mapper_from_near (M,L,b)
13    end
14    P (each n) ← do_map (M)
     // select reducer
15    for each mapper m
16     | k<-get_max_key (m,K,P)
17     | R<-set_reducer (R,k,m)
18     | remove_key (k,K)
19   end
20   O ←do reduce(R)
```

CHAPTER 4. RELATED WORK OF DATA LOCALITY RESEARCH

Chapter 4 illustrates the related works based on the process procedure of MapReduce. The process procedure can be divided into three stages, such as map, shuffle and reduce. Most of researchers increased one of the performance among the three stages. In this chapter, we will explain about the previous research of the data locality in each stage. We will illustrate how the data locality is applied in each stage by chapter 4.

Based on a previous research related to Hadoop performance with big data processing, we introduced data locality in Hadoop and MapReduce. We focused on data locality in software methods without extra cost. So, We did not employ hardware solutions – RAMDISK [88], solid state drives [89,90], or software define network [91] (SDN) – to increase Hadoop performance.

Researchers have tried to minimize network congestion to increase transmission rates, and to increase overall throughput of the system using data locality as a software field that evaluates Hadoop performance. This concept is important to reduce data processing time in Hadoop. It is necessary to understand data locality in Hadoop to apply the concept in Hadoop system. In this section, We divide the three parts of big data processing stages using MapReduce to explain data locality in Hadoop with previous data locality findings.

4.1 Data Locality in Map

Mapper is a container in slave nodes that executes the task with the data block. This is part of the actual data manipulation for the job requested by the client. The map stage works in

parallel with mappers. All mappers in the activated containers are controlled by AM and execute the tasks in parallel. Most study of the map stage has focused on scheduling and increasing Hadoop performance by improving locality. Monitoring using the heartbeat in Hadoop is necessary for task scheduling. Therefore, to improve locality in map stage, We need to know about monitoring and scheduling on YARN. Figure 4-1 shows the relation between master nodes and slave nodes in map stage.

- Monitoring: NM in slave node (Figure 3-3) reports the status of the node – CPU, memory, and disk status – to the RM using heartbeat, the communication method among containers AM and NM. Also, RM stores the information of slave nodes in the metadata. Scheduler performs scheduling based on the information in heartbeat [51] and metadata. This information influences initialization and termination of a job [52]. AM reports a

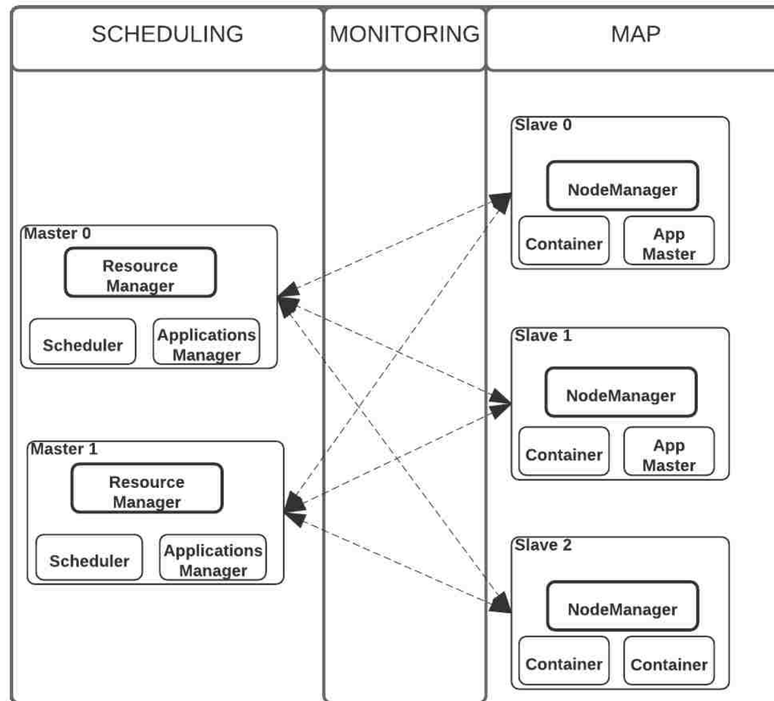


Figure 4-1. Map stage.

hardware and software failure to RM via heartbeat when AM detects the failures from containers and data nodes [53]. Also, RM detects slave nodes that are slower than expected, to be preferentially scheduled on faster nodes. This is called speculative execution in Hadoop. Therefore, the monitoring phase is important to assist scheduler and makes the system highly fault tolerant.

- Scheduling: The scheduler in master node creates several containers in the slave nodes to divide the job into different tasks. NM in slave node is a Java Virtual Machine (JVM) process associated with a collection of physical resources including CPU core, disk, and memory. When a client submits a job to the master node, the RM in the master node executes the AM which decides whether to accept the job or not. Application manager in the master nodes manages the AM that controls the containers in the job. When a master node receives a client's request, the Application Manager selects one AM among the slave nodes to execute the job. The resource managers schedule the job using scheduler if the job is accepted. The scheduler creates several containers in the data nodes based on the Resource Tracker which stores the configure information and status of resources in the data node, to divide the job into several tasks.

Mostly, scheduling tries to allocate the job into slave nodes, that have the input block in the own disk, to reduce rack-local and off-rack using monitoring and heartbeat in Hadoop. Research has focused on scheduling the container (task/mapper) for achieving locality and fairness.

Scheduling research in Hadoop can be divided into three parts like JobTracker, YARN, and Cloud as follows:

1) *JobTracker*: In Hadoop early version 0.X and 1.X, JobTracker in name node manages all slave nodes and applications to process the job. JobTracker distributes the job into slave node through Tasktracker in SlaveNodes in Hadoop version 1.X. JobTracker schedules and handles the entire job. JobTracker directs mapper and reducer to execute the job and selects TaskTracker in slave node. Each TaskTracker in slave node manager directs mappers and reducers to process the MapReduce job. The literature [12 - 22] suggests that scheduling of JobTracker can result in improved data locality in the map stage.
2) *YARN*: After Hadoop version 2.0, YARN is used for scheduling. The Hadoop YARN module is a framework for job scheduling and application management. When YARN has requested the application from the client, it schedules the application into slave nodes. YARN schedules the job based on the slave's status. The literature [54-58] supportsd use of YARN for achieving data locality in the map stage.
3) *Cloud*: Cloud computing provides services, such as Platform as a Service (PaaS) like microsoft Azure [59] or Infrastrucutre as a Service(IaaS) like Amazon Ib Service (AWS) [60]. The provider serves the Hadoop system on a cloud system. The cloud system has a scheduler to allocate the hardware resource to slave nodes in Hadoop. The literature [45-49] suggests use of the cloud scheduler to allocate Hadoop VM into the physically located machine with the same server based on data location.

   Additional studies of data locality in map stage include the following:

4) *Network*: The network status can make a network perspective algorithm [61, 62] employing the queueing model for network routing algorithm with data locality.

5) *Placement*: The literrature [63-69] focuses on location of data and block on HDFS to group the task in the same rack. Studies [70,71] endeavor to gather nodes on the same server by organizing cluster.
6) *Speculative and replica*: When one of the nodes that processes the job with mapper/reducer is down, the other node can replace the node's function as a speculative task with the replicas. The literature considers the speculative [72, 73] and replica [74] to improve locality in the shuffle stage by selecting slave node that already has the block on the local disk in the map stage.

4.2 Data Locality in Shuffle

The key/value pairs in the spilled partition are sent to a reducer in slave node based on the key via the network in the shuffle stage. Most network problems occur in the shuffle stage because of the high data volume. One of the main functions in shuffle stage is data transfer from mapper to reducer. Two more functions in shuffle stage are as follows:

- *Sort/spill*: Before transferring the middle output into reducer, Sort/spill works in mapper as shown in Figure 2-5. The output pair which is emitted by the mapper is called partition. The partition is stored and sorted in the key/value buffer in memory in order to process the batch job. The size of the buffer is configured by Resource

  Tracker in version 1.X (or RM in version 2.X). When its limited is reached, spill is started.
- *Merge*: After reducer receives the partitions from the mapper, the partitions are merged in order to batch job and reduce a stage.

There are mainly three elements to improve locality in shuffle stage (scheduling, block location and key):

1) *Scheduling*: Scheduling in JobTracker [19] or YARN [76, 77] affects locality in shuffle stage like data locality in map. The scheduling in shuffle stage is related to scheduling reducer task and network status. We only count on the scheduling research that considers network status to schedule the reduce task. The other reducer task scheduling will be handled in data localiity in reduce. The literature suggests shuffle service instead of shuffle stage [78] or inside shuffle stage [79] using the scheduler.
2) *Placement*: Hadoop splits input data into several chunks and stores it in the cluster using HDFS. So, block location is related to the locality. There are some off-rack in the shuffle stage if a node in another rack is selected as a reducer. The literature [63, 72, 80] supports reducing the off-rack using block location in HDFS. These studies recommend manipulating block in HDFS before processing MapReduce or application. The block is located in the same rack by grouping blocks [80] or block location policy [63, 72].
3) *Partition/Key*: The partition, middle ouput of the map, is deliverd to the reducer based on the key. So, the key is the main criterion to transfer data into reducer nodes. The literature [81-84, 86, 87] supports reduction of data transferring – selecting the reducer that has most of the key [82, 83], improving the partitioning algorithm [81, 85] in clustering, applying query optimization algorithm [86], using replica [84], pre-shuffle and pre-fetching [87] before shuffle stage.

4.3 Data Locality in Reduce

The reducer in slave node processes the merged partition set complete the application and store the result in HDFS. This is also part of the data manipulation for the job requested by the client, as in map stage. All reducers in the containers activated by AM run the tasks in parallel.

Research on data locality in reduce stage is very similar to study of data locality in shuffle stage because both tried to reduce network traffic between mapper and reducer. Here We divide the research into two parts. One is data locality in the shuffle that focuses on network and partitions in shuffle stage. The other is data locality in reduce stage that concentrates on selecting the reducer in order to improve locality.

Reduce stage can use a scheduler like map stage and service like shuffle stage to improve locality.

1) *Scheduling*: Most scheduling research in reduce stage uses JobTracker, YARN, and cluster like map stage. The research into dynamic scheduling [20, 54, 58] dynamically selects the slot to reduce network traffic using map task location and mapper/reducer information. The previous research of task scheduling [49, 75, 92-96] mostly uses sampling and mapper location to reduce tasks. Also, cluster scheduling [45,46] uses the block and container location in order to remove rack-local and off-local by selecting reducer in the same rack that contains the map tasks.
2) *Service/Framework*: The research [78] suggests a new service between shuffle stage and reduce stage or to make a framework to input an extra module in the middle of MapReduce to improve data locality in the reduce stage [97].

Much work about locality has been completed to reduce off-rack and rack-local. Mostly, locality in Hadoop concentrates on scheduling in MapReduce. Also, the research studies one or

two stages in MapReduce. In the next section, the locality studies are clustered based on this stage, and We show trends in this field.

4.4 Analyzing Previous Data Locality Research

We make a clustering locality research based on the stages in MapReduce, as in Figure4-2. Most research has swarmed in scheduling for improving locality in MapReduce using JobTracker and YARN. Scheduling in reduce stage and shuffle stage correlates to each other, but, We distinguish between scheduling in shuffle stage [19, 76-79] and scheduling in reduce stage [20, 45, 49, 54, 58, 75, 92, 96] based on the network and reducer. Some studies [19, 20, 45, 49, 54, 58, 63] affect two stages in MapReduce to improve locality. Scheduling and data placement overlap in two stages, according to Figure 4-2.

We reorganized the locality studies using Figure 4-2 to analyze trends in locality by creating a timeline similar to Table 4-1. At first, the scheduling in map stage is investigated like traditional scheduling studies. Various data locality impacts in Hadoop appeared after announced

TABLE 4-1. Time line of data locality research

<table>
<tr><th></th><th></th><th>2007 Initial Version 0.14.1</th><th>2009 Version 0.20</th><th>2011 HDFS/MR 0.23</th><th>2012 YARN 2.0-Alpha</th><th>2013 Stable Version 2.2</th><th>2014 REST API 2.5</th><th>2016 3.0 Alpha</th><th>Section</th></tr>
<tr><td rowspan="4">Map</td><td>Scheduling</td><td></td><td colspan="5">[12-22,45-49,54-58]</td><td>N/A</td><td>4.1-1,2,3</td></tr>
<tr><td>Network</td><td></td><td></td><td></td><td></td><td colspan="2">[61,62]</td><td>N/A</td><td>4.1-4</td></tr>
<tr><td>Placement</td><td></td><td></td><td colspan="4">[63-71]</td><td>N/A</td><td>4.1-5</td></tr>
<tr><td>Speculative/Replica</td><td></td><td></td><td colspan="2">[72-74]</td><td></td><td></td><td>N/A</td><td>4.1-6</td></tr>
<tr><td rowspan="3">Shuffle</td><td>Scheduling</td><td></td><td></td><td></td><td colspan="3">[19, 76-79]</td><td>N/A</td><td>4.2-1</td></tr>
<tr><td>Placement</td><td></td><td></td><td></td><td colspan="3">[63,72,80]</td><td>N/A</td><td>4.2-2</td></tr>
<tr><td>Partition/Key</td><td></td><td></td><td colspan="4">[81-87]</td><td>N/A</td><td>4.2-3</td></tr>
<tr><td rowspan="2">Reduce</td><td>Scheduling</td><td></td><td></td><td colspan="4">[20,45,49,54,58,75,92-96]</td><td>N/A</td><td>4.3-1</td></tr>
<tr><td>Service/Framework</td><td></td><td></td><td></td><td colspan="3">[78,97]</td><td>N/A</td><td>4.3-2</td></tr>
</table>

in the HDFS and MapReduce by Apache in 2011. This is relevant to data placement studies because the research can easily modify data location using HDFS. The speculative task research and partition/key research are interested in the performance property of MapReduce, but the researchers are indifferent about the speculative task research because the speculative task is reduced by the stable version of Hadoop.

After adding YARN in Hadoop version 2.0, an interesting network in shuffle and new service in Hadoop became available. With YARN, the developer and researcher are alloId multiple data processing engines, such as real-time streaming, SQL, in-memory, NoSQL, Scala, and so on. YARN works as a data operating system to manage the cluster resource.

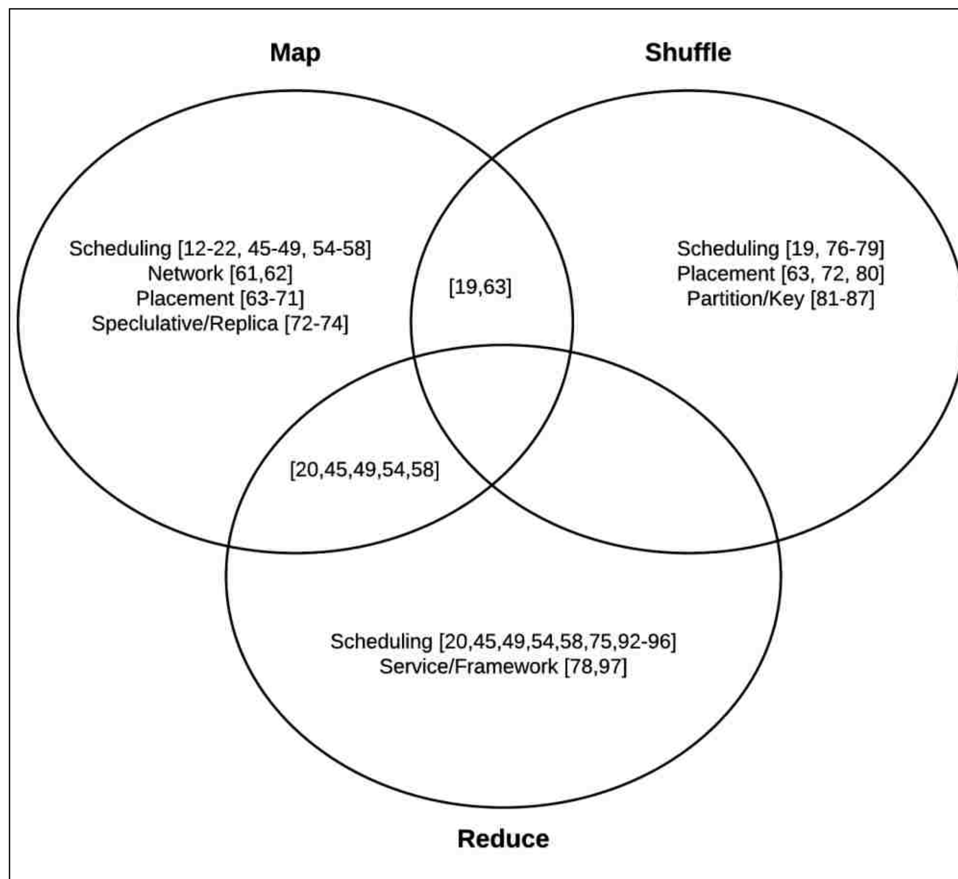


Figure 4-2. Clustering locality research

As a result, the researcher can apply their findings in Hadoop using YARN. Specifically, the shuffle stage begins to increase Hadoop performance using scheduling and data placement. Also, the research of data locality in reduce stage are interested because the researcher can use YARN for increasing Hadoop performance using the framework and new service.

With the upgraded Hadoop, various research areas to improve locality became possible. Data locality measures the performance of MapReduce using various methods, such as Terasort Benchmark [50], Teragen, TeraSort, PageRank, WordCount, and so on. The performance test shows different results based on the type of applications. For big data testing, researchers have to consider data size and type, such as structured, semi-structured, and unstructured. So, We divide the big data application into categories as in Table 4-2, based on a mapper/reducer number and the size of input/middle output. Table 4-2 shows the categories and most efficient research for data locality in Hadoop.

1) *Input > Partition*: Data locality in map stage is most important if input data are larger than middle output (partitions) in shuffle stage. Because the more data is constrated on the mapper than reducer (in the case of the Mapper = Reducer and Mapper < Reducer in Table 4-2). The amount of data in mapper and reducer, However, should be considered

Table 4-2. Category of application type and highest efficiency research

	Mapper>Reducer	**Mapper=Reducer**	**Mapper<Reducer**
Input >Partition	Map, Shuffle, Reduce	Map	Map
Input =Partition	Shuffle, Reduce	Map, Shuffle, Reduce	Map
Input <Partition	Shuffle, Reduce	Shuffle, Reduce	Map, Shuffle, Reduce

when the number of nodes, which role as a Reducer, are fewer than the number of nodes, which role as a Mapper (in the case of *Mapper* > *Reducer* in Table 4-2). Because the reducer could be handle more data than mapper. Work that focuses on improving locality in map stage tries to reduce the size of the data that has to be transferred to other nodes for the map task.

2) *Input* = *Partition*: If the input data is equal to the Partition in size, the mapper/reducer number should be considered. A stage that has fewer taskers (mapper or reducer) should transfer more data than another stage which has more taskers. For example, there are more data trasnsfers in the reduce stage if the number of reducer nodes is fewer than the number of mapper nodes (in the case of *Mapper* > *Reducer* in Table 4-2). In that case, the research that focuses on the shuffle and reduce stage is better performance than the other. If the number of the mapper is smaller than the number of the reducer (Mapper < Reducer in Table 4-2), the research that focuses on map stage is better performance than others.

3) *Input* < *Partition*: If the partition size is larger than the input data, the amount of data transferred is increased in the shuffle stage. Reseachers have tried to reduce data transfer between mapper and reducer using selected nodes that had a mapper, as a reducer. Mostly, the shuffle stage and reduce stage research is better performance than other research when middle output is large (*Mapper* > *Reducer, Mapper* = *Reducer* in Table 4-2). If the number of the mapper is smaller than the number of the reducer (*Mapper* < *Reducer* in Table 4-2), map stage research also could improve the locality for Hadoop performance, because the network traffic can be concentrated on the mapper.

Each locality research study demonstrates different ability based on the application type because each stage in MapReduce uses the different size of data in processing big data.

4.5 Performance Testing

We measured the performance of MapReduce with previous research using Terasort and WordCount to understand the specifics. We divided this testing into two parts with different experimental environments to show various aspects of data locality.

4.5.1 Terasort Benchmark

Terasort Benchmark [50] is commonly used for measuring the performance of MapReduce [66-68]. We conducted testing on CloudLab [98], a cloud computing platform for research. My test utilized one master node which manages HDFS, one RM node, and ten slave nodes. Each node was equipped with 6 gigabytes memory, two Xeon E5-2650v2 processors (eight cores each, 2.6 GHz) and 1 TB hard drive. Hadoop 2.7.1 was installed on a Linux machine

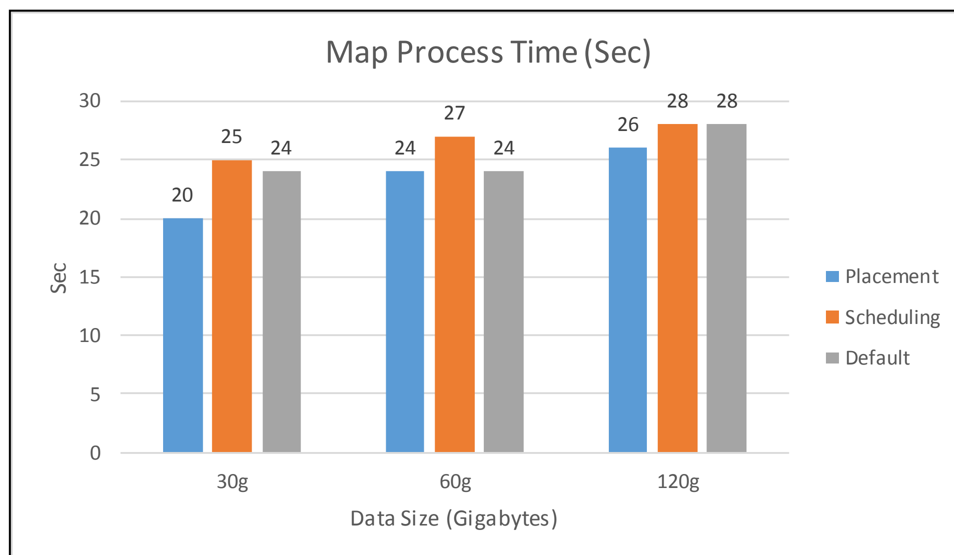


Figure 4-3. Process time of Map in Terasort.

running Ubuntu 14. Six different random data sets were created using Teragen (i.e., two 30 GB data, two 60 GB data, and two 120 GB data). The data consisted of key and value pairs. The data set sizes were chosen following research on Hadoop testing by Intel [99].

We used ten mappers and one reducer to process the data. Figure 4-3 shows the result of Map processing with default MapReduce, data locality scheduling, and data placement. The data placement increased the data-local in map stage by placing the data in the nodes, which will work as a mapper. The placement worked when the input data was uploaded on HDFS before processing MapReduce. Some extra processing time might be required when uploading the input data into HDFS. The scheduling considered all angles of MapReduce with any given situation. Sometimes the processing time in map stage was increased because of rack-local in data locality scheduling.

In shuffle stage, However, the processing time with the scheduling took less time than

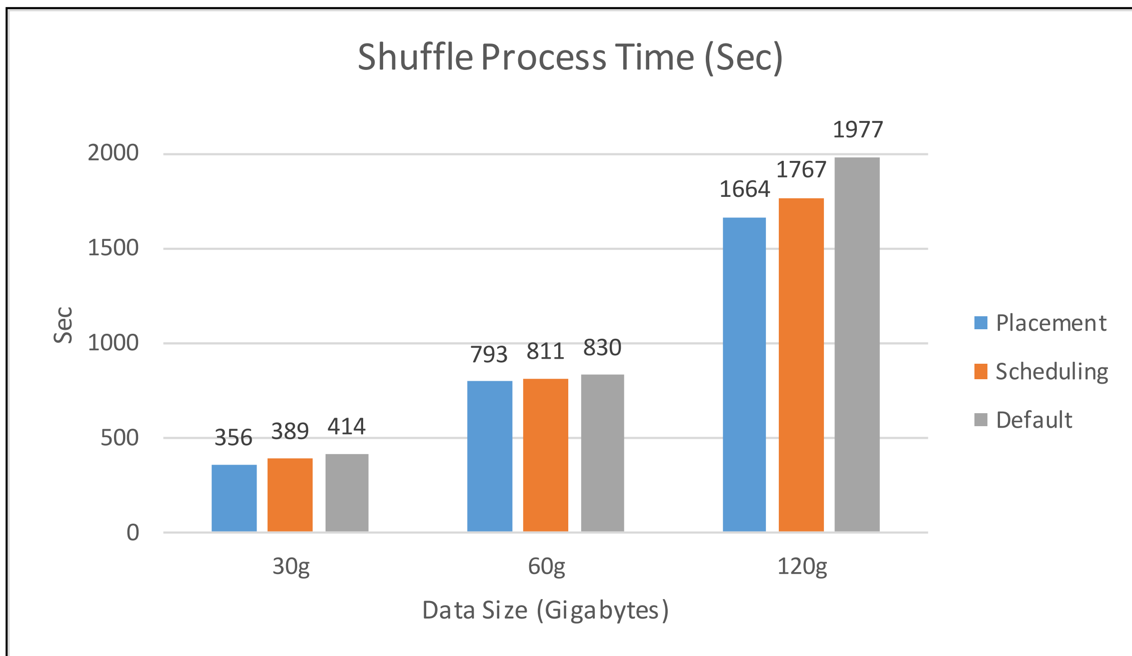


Figure 4-4. Process time of shuffle.

default processing time in shuffle stage. Figure 4-4 shows the result of processing time in shuffle stage. Data locality scheduling took less time (about 200 sec) than default MapReduce in Figure 4-4 with 120 gigabytes of data. Data placement and data locality scheduling reduced the processing time in shuffle by reducing the rack-local and off-rack. The Terasort application is in *Input* = *Partition* and *Mapper* > *Reducer* in Table 4-2. So, the shuffle stage is more efficient than map stage to increase performance of MapReduce.

4.5.2 WordCount

In the previous test, We used large-scale data set sizes (30 gigabytes, 60 gigabytes, and 120 gigabytes), but We was limited by the network and physical disk location in Cloud Lab. So, We installed Hadoop 2.7.1 in five physical machines in the University of Nevada, Las Vegas (UNLV) Security Lab. The master node and RM node were installed in the machine with 16

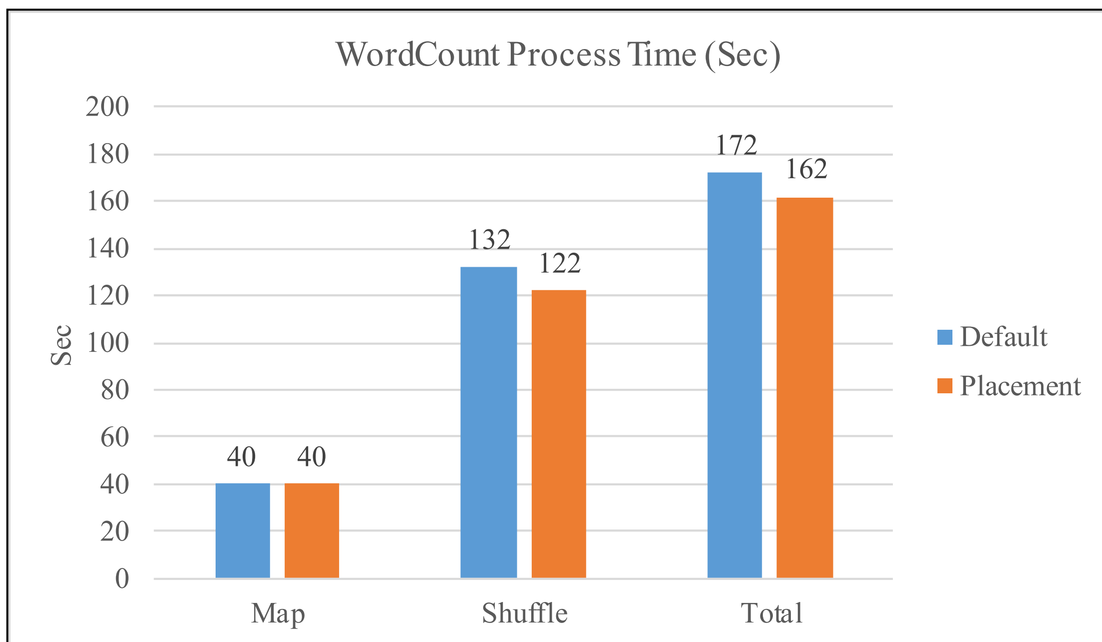


Figure 4-5. WordCount process time with default and data placement.

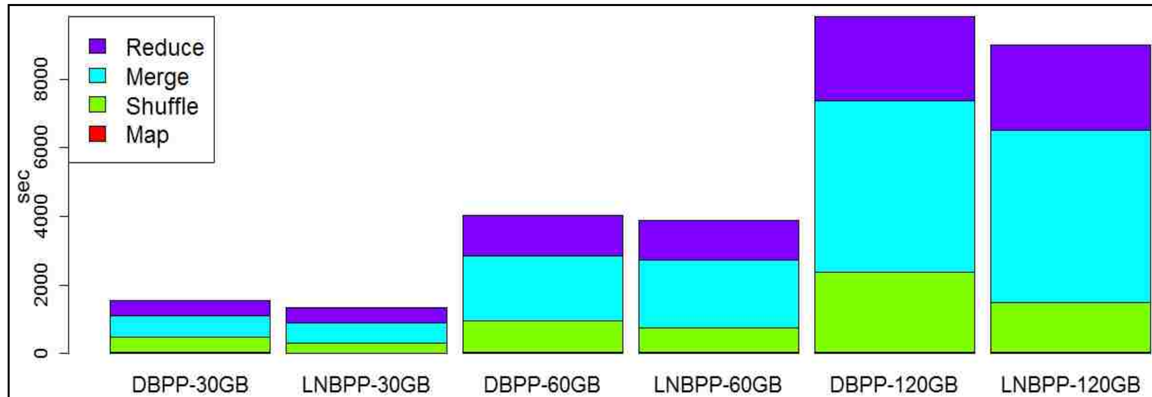
gigabytes memory, Intel Core i5, and solid state drive. The four slave nodes were installed in the machines with 8 gigabytes memory, Intel Core i5, and the eMMC disk. We used 1 gigabyte random data for MapReduce with WordCount.

The process time of reduce stage was 0 because the output of reduce stage in WordCount got too smaller after shuffle stage, also, already processed in sort/spill and merge phase. The result of processing time is shown in Figure 4-5. In this test, We only compared between MapReduce on the default Hadoop system and MapReduce with data placement. There was no rack-local or off-rack in the map stage because We used only four slave nodes with three replicas in the same switch. There was network traffic in the shuffle stage because We used sixteen mappers and two reducers. The partitions in the sixteen mappers had to be transferred to reducers.

We only applied simple data placement on HDFS. Data placement on HDFS can reduce the processing time of the shuffle stage by selecting the reducer among the nodes that has most of the partitions. In the WordCount test, the processing time was not influenced much by data locality in the shuffle stage and reduce stage because the middle output was small and the number of reducers could be increased. This test was still valuable, However, when the input data and middle output are large, as in Figure 4-4.

4.6 Limitation of Shallow Data Locality

While the data locality works in small scale, it does not work as expected in large scale where Hadoop is most needed. There are two major reasons for this. First, when there are a large number of data blocks and computing nodes, there is inefficiency on block assignments, and Hadoop will attempt to utilize the computing nodes even when the data block is not locally present. The number of RLMs also gets increased when the number of blocks is increased.



*Map appears as a line as it is too small

Figure 4-6. Processing Times of Each Step of MapReduce.

Second, the mostly data locality currently only applies to one or two stages - map, shuffle and reduce stage. We have analyzed the processing times of each stage with Terasort Benchmark [50] and obtained the result in Figure 4-6, which clearly shows that the later stages take much longer time than Map in the Terasort Benchmark test. That means Shuffle and Merge in some case of application takes longer than other stages such as reduce and map.

Unfortunately, the majority of the big data processing cost is actually in the later stages (shuffle, merge, reduce) in some application case. Current most research of data locality have focused only on one stage with specific application type (one of Table 4-2). They overlook the other stages such as map, shuffle and reduce stage. Even though the Rack-Local and Off-Rack increase in Reduce stage like Figure 4-7. The data locality on Hadoop can change based on the stage of MapReduce and the type of application.

Therefore, the data locality in the all stages such as map, shuffle, reduce, have to be consider to improve performance of Hadoop processing time.

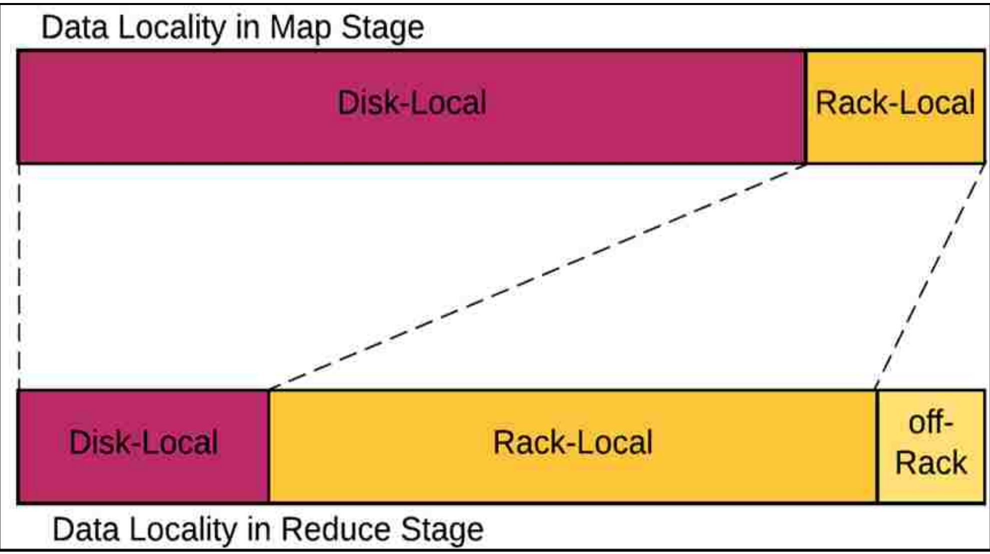


Figure 4-7 Changing Data Locality between Map and Reduce step.

CHAPTER 5. MAPREDUCE COMPUTATIONAL MODEL

Chapter 5 explains the Hadoop performance analysis model. In this chapter, the simple computational model is illustrated for understanding the process of MapReduce. We advanced the simple computation model by adding the data locality. The advanced Hadoop performance model will be explained in 5.2 section. The 5.2 section will show the advantage about the data locality of Hadoop performance by comparing it to the default MapReduce. By the section, we can calculate the performance of Hadoop with data locality.

5.1 Simple Computational Model

We developed a simple computational model to understand the performance of MapReduce based on Terasort in its default configuration. The model does not consider other factors, such as in-memory computing [39, 40, 41], algorithms [100], scheduling [27,31], etc. This model considers only four major stages, i.e., map, shuffle, merge and reduce. We do not consider three small stages of, scheduling, monitoring, and output. They are rather related to HDFS and their stages are mostly controlled by the HDFS configurations. We merge the Spill/Sort stage into the map stage because the spill/sort is invoked only when the mapper makes an output. Each container processes one block at a time. We assume that the processing time (P) of a container with one block is identical for all containers. And, all containers start the job at the same time.

1) *Map*: Its performance depends on the number of containers ($|C|$) and the number of Blocks (B). Each container can get a different number of blocks. We define sets C and B as following.

$C = \{C_1, C_2, \ldots C_i \ldots C_k\}$, where C_i is a Container

$B = \{B_1, B_2, \ldots B_i \ldots B_k\}$, where B_i is the number of blocks in C_i.

The containers work in parallel in the Map stage. So, the total processing time for the Map stage (T_M) is defined as follows.

$$T_M = Max\ \{P \times B_j\}, \quad where\ j = 1 \ldots k$$

2) Shuffle: The total process time of Shuffle stage depends on the amount of mapper's outputs. There are several methods of data delivery, but We employ a FIFO method for simplicity. The delivered file size depends on the mapper's output, which is same as the number of blocks in case of Terasort. We define F as a set of numbers of delivered files in a container:

$F = \{F_1, F_2 \ldots F_j \ldots, F_n\}$, where F_j is a number of delivered files from C_j.

The total number of files sent from mapper to reducer via shuffle is:

$$F_T = \sum_{i=1}^{n} F_i$$

Let t_f be the delivery time of one file. Then the total processing time (T_s) of Shuffle stage is:

$$T_S = F_T \times t_f$$

3) Merge: Merge stage is started when the output of mapper reaches a slave node which works as a reducer. We use one reducer in our Terasort Benchmark. For simplicity, we

assume that all mapper's output is received by one reducer, although multiple reducers working in parallel can be used. The processing time of Merge stage depends on how many mapper's output is made. The total number of mapper's output is:

$$B_T = \sum_{i=1}^{k} B_i$$

Let t_{merge} be the merging time between B_i and B_{i+1}, then the total processing time (T_E) of merge is:

$$T_E = (B_T - 1) * t_{merge}$$

4) Reduce: When Merge stage is finished, the container works as a reducer. In the Terasort Benchmark, only one slave node is used as a reducer. So, the processing speed depends on the resource of the selected slave node and the size of the input data. Like in mapper, We define P as the processing time of one block in the container. Then the total processing time (T_R) of reduce is:

$$T_R = P \times B_T$$

By combining all the above, We get the total time (T_P) of MapReduce processing:

$$T_P = T_M + T_S + T_E + T_R$$

Note that the number of blocks such as B_T cannot be controlled directly by MapReduce because it is determined by the configuration of the HDFS which sets up the block size and the input data size. Also, there is an inverse relationship between B_T and P. For example, if B_T is reduced in the configuration of HDFS, then P is increased because the block size gets larger.

Therefore, researchers have tried to reduce T_M and P using scheduling [29], hardware solution [101], in-memory computing [39,40,41], data locality [28, 29], and so on. Also, they have tried to reduce t_f using RESTful API, Software Defined Network [36] and Remote Direct Memory Access (RDMA) [37], and so on.

5.2 Advanced Analyzing Hadoop Performance Model

In the previous section, we made the simple computational model based on MapReduce process. However, the simple computation model is not enough to measure the effectiveness of data locality in Hadoop. Data locality is related on the location of blocks in HDFS and the location of keys in blocks. Therefore, we should need to combine the simple computational model with the location information. We made a more detailed computational model for deep data locality by including the keys and location information into computation model. We explained the computational model for deep data locality in the section stage by stage.

Table 5-1. Constants

Constant Symbol	Definition
α	Processing time for one block
β	Transferring time of one block under Rack-Local
δ	Transferring time of one block under Off-Rack
$\boldsymbol{\gamma}$	Processing time of Sort, Spill and Fetch for one block

Table 5-2. Time Variables

Time Symbol	Definition
T_1	Processing time of First Stage ($T_M + T_S$)
T_M	Processing time of Map function
T_S	Processing time of Sort, Spill, Fetch in Shuffle
T_2	Processing time of Second Stage ($T_T + T_R$)
T_T	Processing time of Transfer in Shuffle
T_R	Processing time of Reduce function
T	Total processing time of Hadoop

Table 5-3. Other Variables

Symbol	Definition	Symbol	Definition
M	Number of Mapper in Map	P_r	Ratio of Partition in Mapper, $\{P_1, P_2, \ldots, P_r\}$
R	Number of Reducer in Reduce	$\|B_i\|$	Total number of allocated blocks in Mapper (i)
RLM_i	Number of Rack Local Map (RLM) in Mapper (i)	B_i	Set of allocated blocks in Mapper (i), $\{ b_1, b_2, \ldots, b_B\}$
i	Mapper ID	R_{DL}	Ratio of Disk-Local
j	Reducer ID	R_{RL}	Ratio of Rack-Local
		R_{OR}	Ratio of Off-Rack

How important is data locality and how much improvement can DDL bring in? To answer it, we have developed a Hadoop performance analysis model. This model is divided into two stages, 1) Map function and sort/spill/fetch, processed by the mapper, and 2) Transfer and reduce function, processed by the reducer. Table 5-1, 5-2, and 5-3 summarizes the notations used in this model. Figure 5-1 illustrates the related stages for the time variables.

5.2.1 First Stage (T_1)

All mappers work in parallel to make partitions from the allocated blocks. The resulting

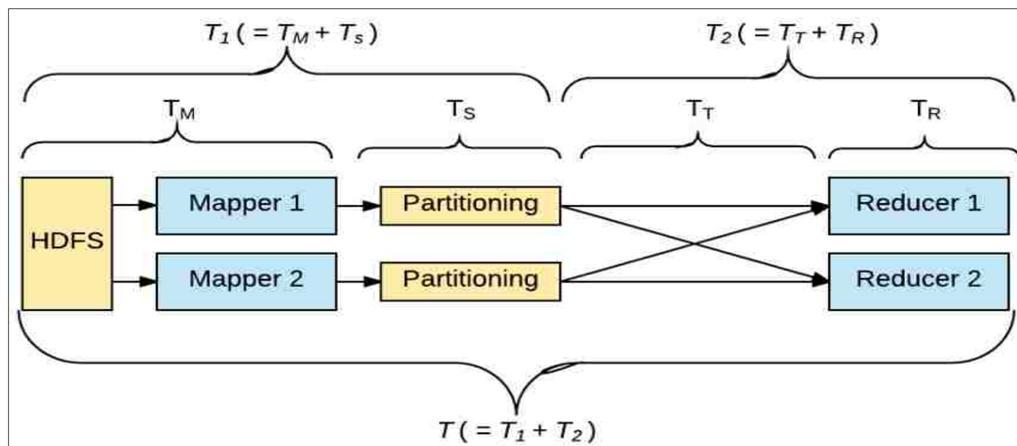


Figure 5-1 Time Variables for Each Stage

partitions go through sort/spill and fetch. There are two data localities at this stage, i.e., Data-Local and Rack-Local. The processing time of each block takes the same time (α) in mapper (i). Transferring each block (b) takes the same time (β). Therefore, we have:

$$T_M\ (i,b\ ) = \begin{cases} \alpha\,, & (If\ block\ is\ in\ the\ Node) \\ \alpha +\ \beta\,, & (If\ block\ not\ in\ the\ Node) \end{cases} \quad b\ is\ a\ block\ id$$

Figure 5-2 shows the map function time in mapper (i). The processing time of Map function (T_M) depends on the number of allocated blocks and the number of RLM:

$$T_M(i)=\{\ |B_i|\ \ * \alpha + RLM_i * \beta\}$$

The processing time of sort/spill and fetch on each mapper can be calculated by multiplying the number of allocated blocks and processing time of Sort, Spill and Fetch for one block (γ):

$$T_S(i) = (|B_i| * \gamma)$$

The processing time of the first stage on each mapper can be calculated by adding the above two

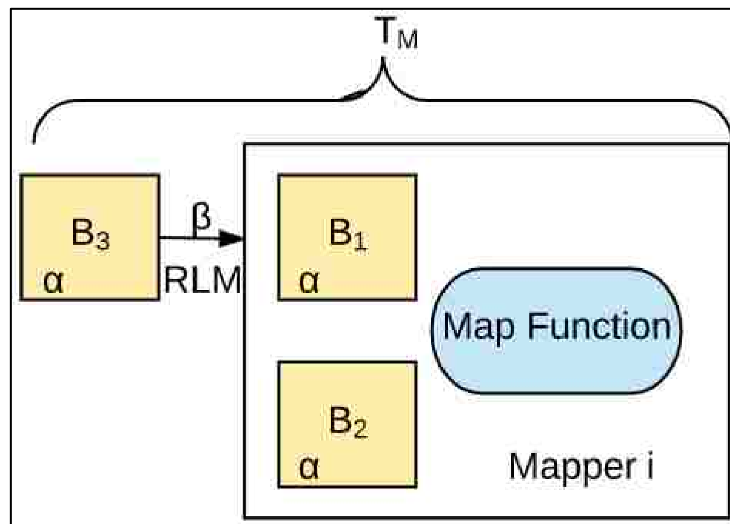


Figure 5-2 Map Function time.

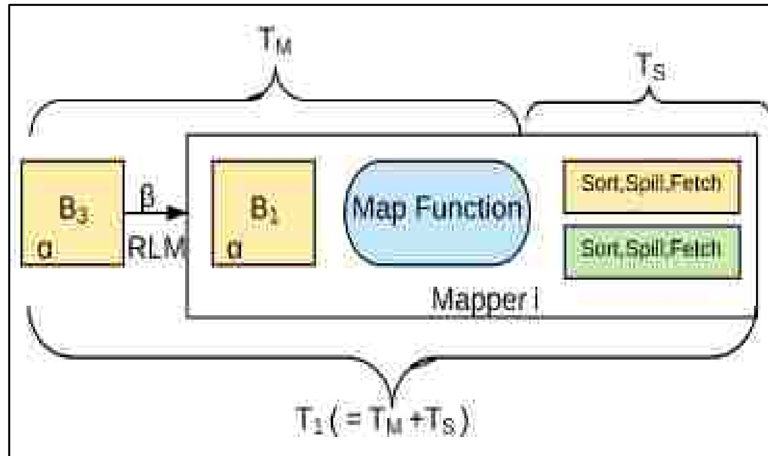


Figure 5-3 Processing Time of First Stage on Mapper (i).

processing times as shown in Figure 5-3. The longest processing time among all mappers is the processing time of the first stage (T_1):

$$T_1 = \max_{1 \le i \le M} \{T_M(i) + T_S(i)\}$$

$$= \max_{1 \le i \le M} \{ |B_i| * \alpha + RLM_i * \beta + (|B_i| * \gamma)\}$$

$$= \max_{1 \le i \le M} \{ |B_i| * (\alpha + \gamma) + RLM_i * \beta\}$$

Figure 5-4 shows the effect of RLM in this model. The more RLM blocks there are, the longer the processing time becomes.

5.2.2 Second Stage (T_2)

This stage includes two processing times, that is, the transfer time between mapper and reducer, and the processing time of reducer function. To get them, we need to know the middle output size on the reducer (*j*). The middle output size on the reducer can be calculated by the

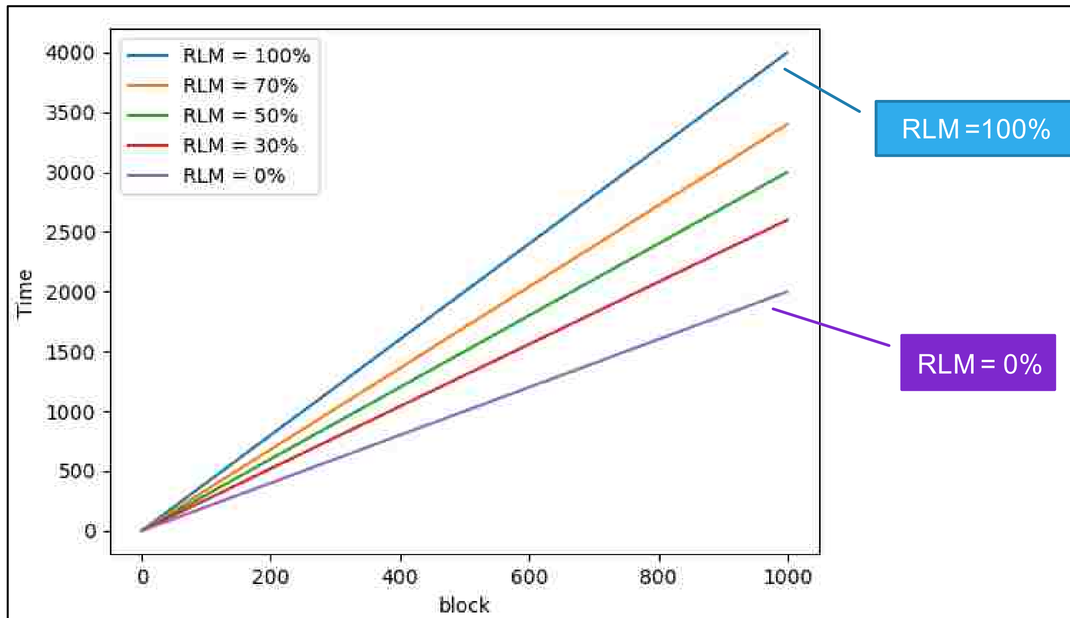


Figure 5-4 First Stage Calculation Graph.

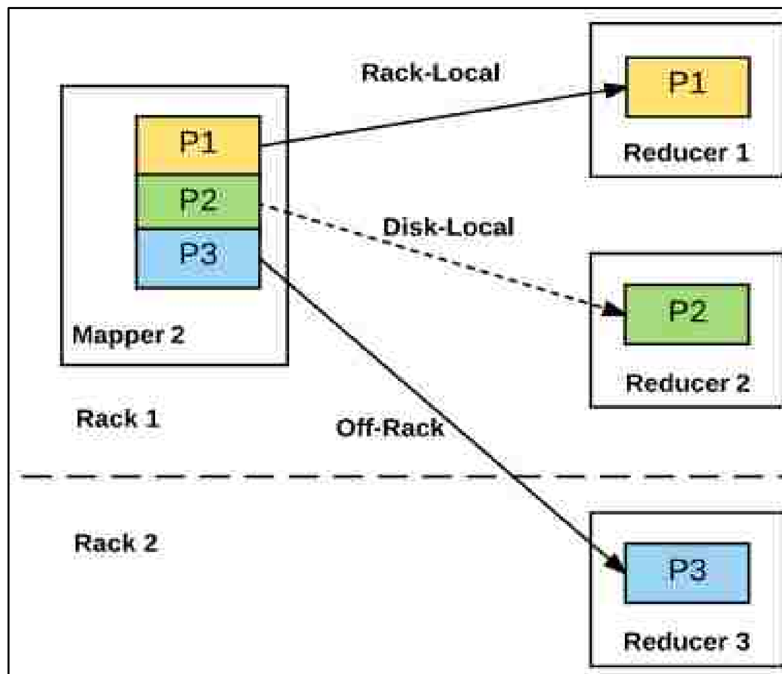


Figure 5-5 Data Locality in Second Stage.

partitions in the mappers because each mapper is supposed to send the partition to the reducer.

The middle output size of the reducer is the total number of block multiplied by the partition ratio per block:

$$P_r(j) * \sum_{i=1}^{M} |B_i|$$

In the second stage, there are three data locality types, *i.e.,* disk-local, rack-local and off-rack (Figure 5-5). Each partition on mapper has one of those locality types. For example, if the partition is supposed to stay in the same node, it is Data-local. If the partition is supposed to be transferred to a different node in the same rack, it is rack-local, otherwise, it is off-rack. Using the transfer time of one block under rack-local (β) and off-rack (δ), the network transfer time between mappers and reducer (T_T) can be calculated as (Figure 5-6):

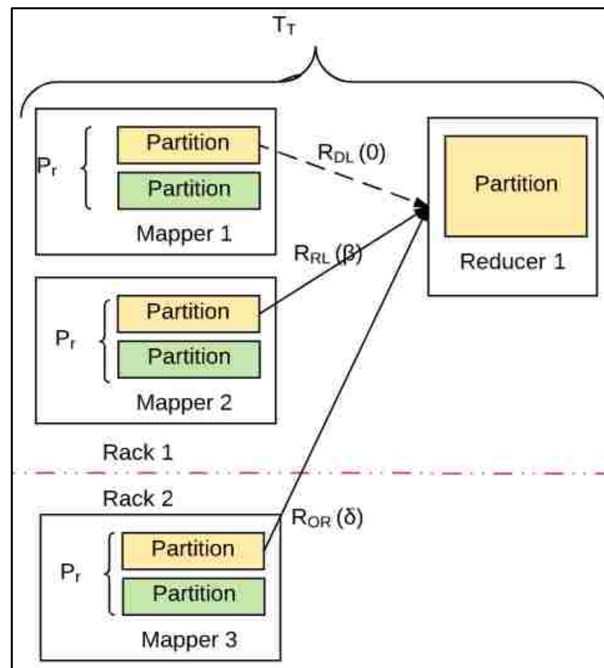


Figure 5-6 Transfer Time (T_T) to reducer (j)

$$T_T(j) = P_r(j) * \sum_{i=1}^{M} |B_i| * \{0 * R_{DL}(j) + \beta * R_{RL}(j) + \delta * R_{OR}(j)\}$$

$$= P_r(j) * \sum_{i=1}^{M} |B_i| * \{\beta * R_{RL}(j) + \delta * R_{OR}(j)\}$$

The transfer time (T_T) under disk-local is 0 because the partition is already in the reducer. The processing time of reducer function depends on the size of the middle output. So, the processing time of reduce function is the size of middle output on reducer multiplied by the processing time for one block (α):

$$T_R(j) = P_r(j) * \sum_{i=1}^{N} (|B_i| * \alpha)$$

Figure 5-7 shows the process of the second stage on the reducer (*j*). It is the longest processing time among the reducers, which is equal to the max value among the transfer time plus the processing time of reducer function:

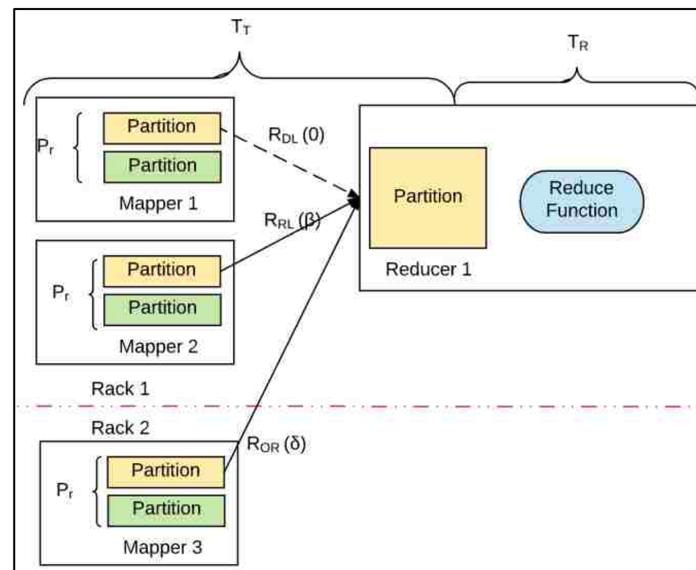


Figure 5-7 The Process of the Second State (T_2) on the Reducer (*j*).

$$T_2 = \max_{1 \le j \le R}\{T_T(j) + T_R(j)\}$$

$$= \max_{1 \le j \le R}\{P_r(j) * \sum_{i=1}^{N}|B_i| * \{\beta * R_{RL}(j) + \delta * R_{OR}(j)\} + P_r(j) * \sum_{i=1}^{N}(|B_i| * \alpha)\}$$

$$= \max_{1 \le j \le R}[P_r(j) * \sum_{i=1}^{N}|B_i| * \{\alpha + \beta * R_{RL}(j) + \delta * R_{OR}(j)\}]$$

Figure 5-8 shows the second stage processing time under different rack-local and off-rack ratio. As the ratio of either rack-local or off-rack grows, the processing time increases, too. Especially, the off-rack blocks have a greater impact in increasing the processing time.

5.2.3 Total Hadoop Processing Time (*T*)

The total Hadoop processing time can be obtained by adding two stages:

$$T = T_1 + T_2$$

$$= \max_{1 \le i \le M}\{|B_i| * (\alpha + \gamma) + RLM_i * \beta\} + \max_{1 \le j \le R}[P_r(j) * \sum_{i=1}^{N}|B_i| * \{\alpha + \beta * R_{RL}(j) + \delta * R_{OR}(j)\}]$$

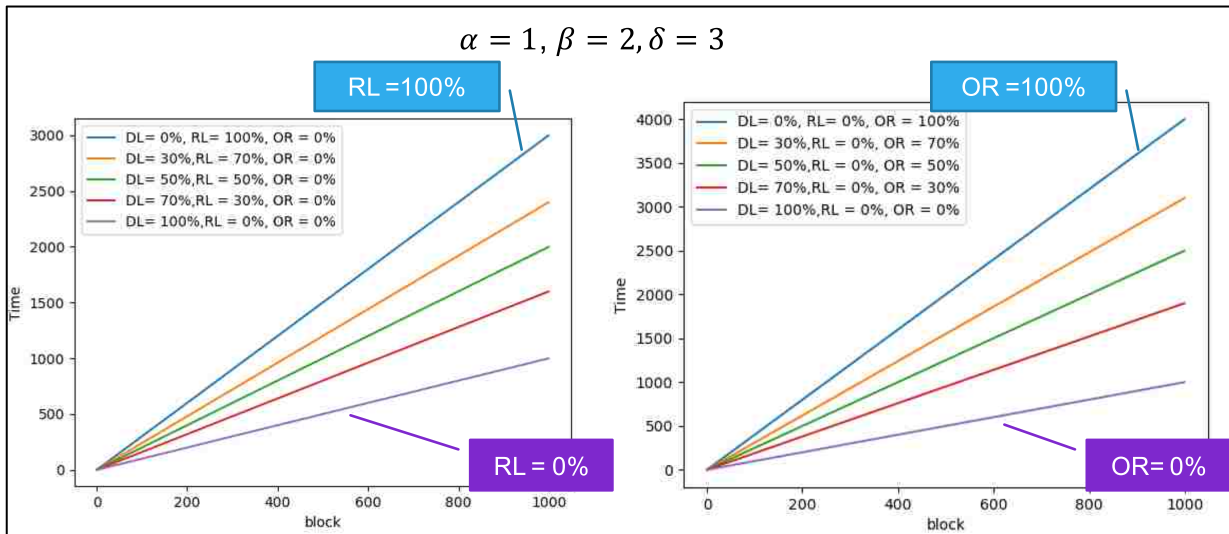


Figure 5-8 Second Stage Calculation Graph (T_2).

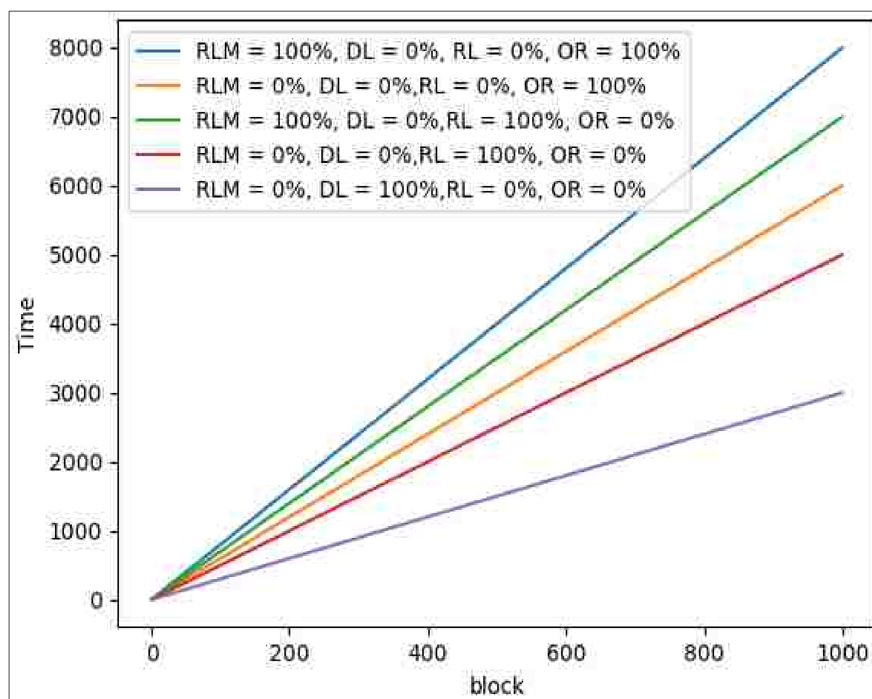


Figure 5-9 Total Hadoop Processing Time Calculation Graph.

Figure 5-9 shows the graph of total Hadoop processing time. Rack Local Map (RLM) in the first stage, and rack-local (R_{RL}) and off-rack (R_{OR}) in the second stage are the locality types that negatively affect the Hadoop performance. Compared with the ideal case where the disk local is 100% in both stages, when the

RLM is 100% in the first stage and the rack-local is 100% in the second stage, total Hadoop processing time is twice larger. Furthermore, if the second stage has 100% off-rack data, the total Hadoop processing time is three times larger than the case of data-local only. This observation gives an important insight in data locality. DDL minimizes the RLM on the first stage and maximizes the data-local on the second stage, thereby increasing the performance of Hadoop.

In this chapter, we showed the computational model for deep data locality on Hadoop. By reducing the RLM in the first stage and rack-local and off-rack in the second stage, we improved the performance of Hadoop theoretically. In chapter 6, we will show that how to implement of DDL in the real Hadoop system and will prove the performance improve via cloud test, physical test, and simulation (Appendix A).

CHAPTER 6. DEEP DATA LOCALITY

In this chapter, we will explain how to apply the DDL concept into Hadoop system via the DDL methods. While the data locality is very important in Hadoop, it has been applied only to the early stages of Hadoop processing. In this section, we elaborate on the concept of DDL (Deep Data Locality) that organizes the data in a way to minimize the data transfer, which reduces the overall processing time.

Hadoop places the data blocks on the associated processing nodes as much as possible to reduce the amount of data transfer. This concept is called data locality, but it has been utilized only in the early stages while the majority of the processing costs occur in the later stages. Therefore, the benefit of the data locality has been limited. We extended the data locality concept to the later stages by pre-arranging the data blocks to reduce the data transfer during the later stages as well as in the early stages. We will call this new concept Deep Data Locality (DDL) and refer the conventional scheme as Shallow Data Locality (SDL). Deep data locality concept has not been studied before, yet it has a great potential to improve the Hadoop processing performance without any extra hardware cost.

One of the concept of Deep Data Locality is "Good Writing makes a Good reading". Hadoop also can apply the concept into block orgaining and re-arrange inside of the block to improve the performance of analyzing speed in the Hadoop system.

6.1 The Concept of Deep Data Locality

Achieving data locality at map stage is quite intuitive. The input data is split into smaller data blocks and distributed to slave nodes. Each slave nodes just need to process the data blocks copied to its hard drive. We refer this type of data locality SDL (Shallow Data Locality). However, the data gets relocated in shuffle stage and the reduce stage cannot utilize any locality. If the data is pre-distributed to the appropriate reducer nodes, it will reduce the shuffle time and improve the performance. We refer this type of data locality DDL (Deep Data Locality). To fully utilize DDL, an integrated approach between Hadoop and MR is needed.

The Deep Data Locality (DDL) can be divided two parts. One is block-based DDL which is manage the block location in data nodes to improve performance of MapReduce. The block-based DDL will be dealt with next section with detail example of LNBPP. Second part is key-based DDL with ETL. The key-based DDL is fine-grain manage than block-based DDL. So, the key-based DDL has to consider starting from big-ETL. The key-based DDL with ETL will be discused in the key-based DDL section and ETL-D (Extract, Transfer, Load with DDL).

6.2 Block-based DDL

While DDL can be promoted in a number of different ways, We study the block placement algorithms as the first stage in this research. DDL based on block manage the block in HDFS to improve the performance of processing applications in YARN. There are several block-based DDL methods. As a preliminary stage towards block-based DDL, We designed a new block placement algorithm called limited node block placement policy (LNBPP). LNBPP algorithm replicates the data blocks to the nodes that would become Reduce nodes during the Hadoop block replication process, i.e., before the MapReduce process begins.

6.2.1 LNBPP

My proposed algorithm LNBPP reduces the number of RLMs to improve the performance of map, and reduces the numbers of delivered files (F) in a container to increase the performance of shuffle (Chapter 5). A MapReduce process with LNBPP has a more consistent performance across slave nodes because of there is no fluctuation in number of RLM. As a result, the block assignment by the LNBPP improves the overall performance of Hadoop system. In this section, We describe the LNBPP mechanism in detail.

The container in the MapReduce is allocated by the data locality and the status of resources in data node. The previous MapReduce researches have tried to make balanced workload in slave nodes using data location [27, 29], scheduling [31] and so on, in order to achieve the best performance, but without considering the mapper's output and reducer's input. In our tests, it was shown that the Shuffle take a big portion of the MapReduce processing as illustrated in chapter 4's Figure 4-7. It shows the processing times of each stage in MapReduce. Both DBPP and LNBPP make multiple replicas of the blocks in HDFS to assure the reliability. We have examined the method of decreasing data transfer time in order to decrease the shuffle time.

During this process, LNBPP organizes a group of blocks (or containers) for a job in a way to minimize the shuffling by strategically positioning the replicas. If a replica is located on the same node with other containers that would need the replica later, no shuffling would be needed. In short, the basic concept of LNBPP is reducing the time of shuffle by removing or reducing the number of delivered partitions in a container. We have examined the method of decreasing F_T in order to decrease the shuffle time. The basic concept of LNBPP is reducing the time of shuffle by removing or reducing the number of delivered files (F) in a container.

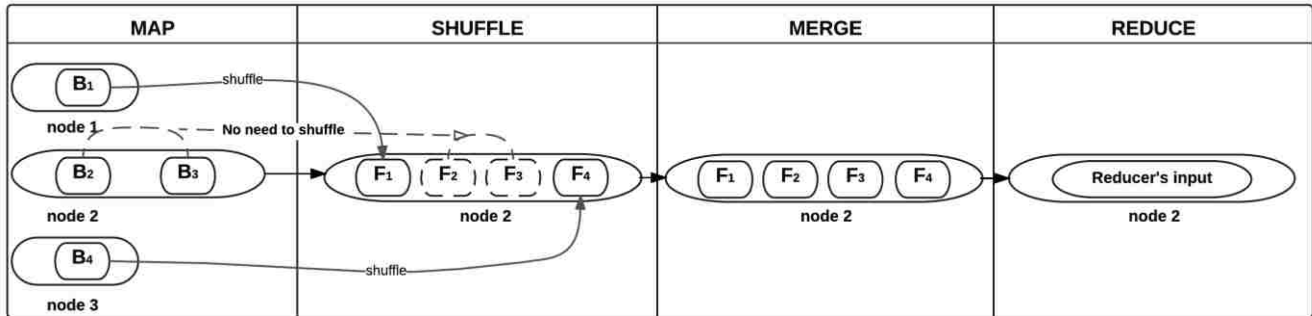
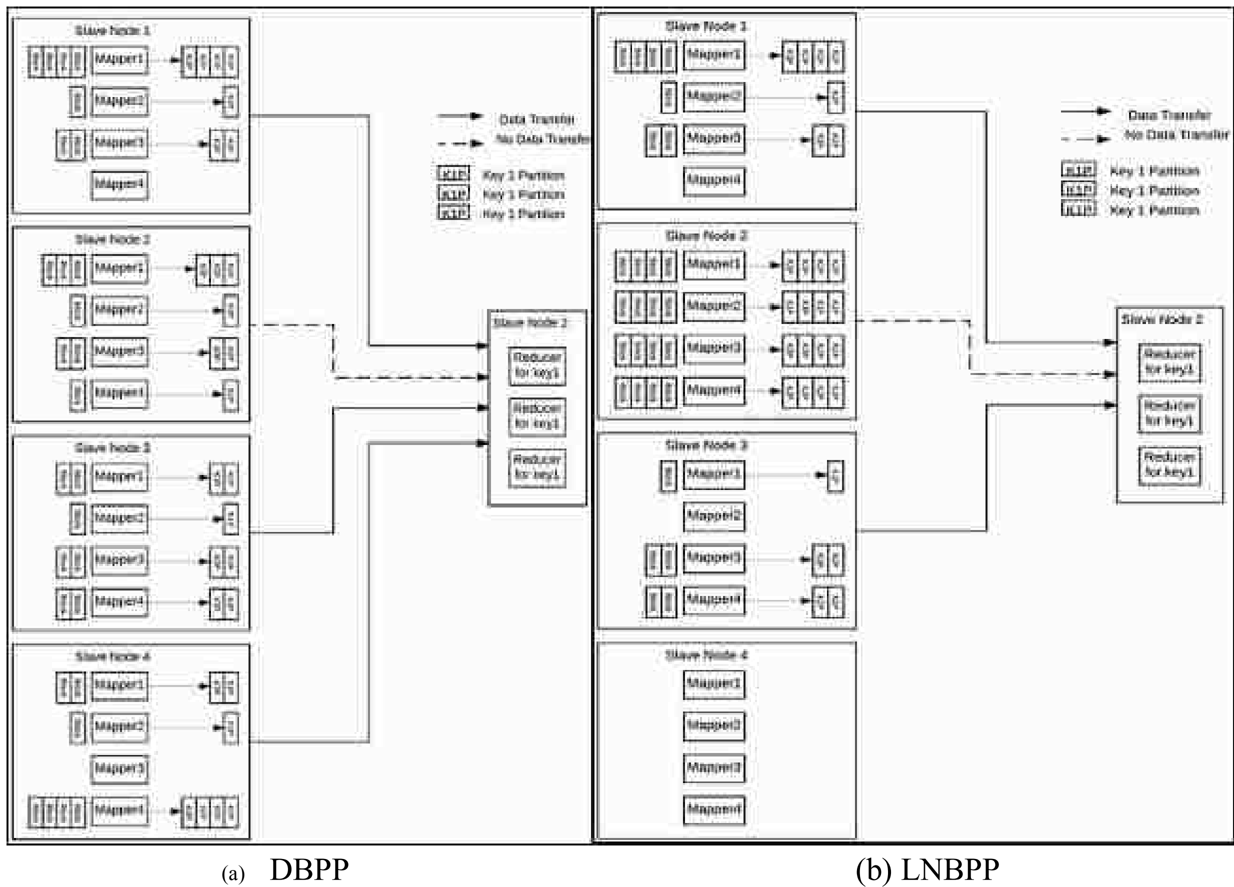


Figure 6-1. Decreased Shuffle time by LNBPP.

Figure 6-1 illustrates the process of MapReduce and explains how LNBPP can save the execution time intuitively. In Map stage, the blocks are distributed to nodes. A node can contain multiple blocks. For example, the second node contains B_2 and B_3. The blocks are converted into files, F_1, F_2, etc. In the shuffle stage, the files need to be transported to the reducer node, consuming network bandwidth and causing a delay. By using the node with the most files as the reducer, this traffic can be minimized. In this example, the 2nd node containing B_2 and B_3 is chosen as the reducer.

Moreover, locating an additional replica in the same node does not increase processing time as long as the node has an unutilized core. As a result, a group gets to occupy a minimal number of slave nodes to in order to reduce data transfer time. This reduces the number of inter-node shuffling of a file. Finally, LNBPP selects the container with the most number of blocks in the group as a reducer. This causes utilizing a reduced number of nodes rather than all available slave nodes, hence “Limited Node” BPP.



(a) DBPP (b) LNBPP

Figure 6-2. Block locations in MapReduce.

The detail example is illustrated in Figure 6-2. With the default policy shown in Figure 6-2 (a), the activated containers are evenly spread out to keep the balance. In this case, any node could become a reducer, when mapper finishes the job. The slave node that becomes reducer must collect the rest of mapper's output from other slave nodes in order to perform reduce stage. For example, in case of the Figure6-2 (a), the Slave Node 1, 2, 3 and 4 have same number of blocks (B) to process jobs in Map Stages. After finishing the Map Stage, Slave Node 2 is selected as a reducer in Reduce Stage by Applications Manager in YARN. Therefore, Slave Node 1,3 and 4 have to send their partitions (P) to Slave Node 2. In the Figure 6-2 (a), Slave

Node 2 receive 21 partitions from other slave nodes via network. The 21 partitions cause network traffic base on the size of the partition.

The MapReduce on the LNBPP has a different block location as shown in Figure 6-2 (b). The nodes which will be reducers, process blocks as much as possible using their Mapper in Map Stage. By selecting the nodes which process most of block in Map Stage, the transfer time among the nodes are reduced. For example, in case of Figure 6-2 (b), Slave Node 2 process most of block as best Mappers in Slave Node 2 can execute. After finish Map stage, Slave Node 2 is selected as a reducer in Reduce stage. Therefore, the reducer keeps 16 partitions (P) in Slave Node 2, and, receive only 12 partitions (P) from Slave Node 1 and Slave Node 2. Shuffle stage in Figure 6-2 (b) is 9 partitions (33%) less data transfer comparison with Shuffle stage in Figure 6-2 (a).

Algorithm 6-1 shows how to make a group of nodes for LNBPP, i.e., LN (Limited Node) group. The algorithm accepts the list of block numbers (D) from each nodes as an input. In the first while loop, LNBPP searches for the node which has the largest number of blocks in D to make it reducer (R), and make a list of nodes (L) which has those blocks. Then L is sorted by the number of blocks in each node and R is added to the LN group (G). In the second while loop, the members of LN group (G) is updated by adding a node from L continually until the nodes in G includes all the blocks in D. In the third while loop, L is updated by removing the block numbers which is already in G.

This not only reduces the amount of shuffle, but also has an effect of eliminating the RLM. After executing Algorithm 6-1, the member nodes in LN group include all input blocks for the next Map stages, so there is no need to copy any missing block, which eliminates RLM.

Through LNBPP, the degree of the data locality during Shuffle stage is increased, resulting in stronger DDL.

Algorithm 6-1: Make a LN Group on LNBPP

```
     Input: D (the list of block number in the data)
     Output: G (LN group), R (Reducer ID number)
    // b: list of block numbers, L: list of nodes and b, N: node
1    Initialization: G ← 0, R ← 0, b ← 0, N ← 0, L ← (0, 0)
2    // This creates the list of blocks for each node
    while N ← get_next_node () < > 0 do
3     |  b ←  get_number_blocks (N, D)
4     |  if R < b then
5     |  | R ← N
6     |  end
7     |  if b > 0 then
8     |  | add_node_L (N, b)
9     |  end
10   end
11   sort_L ()
12   update_node_G (R)
13   // This creates LN
    while N ← get_next_node_L () < > 0 do
14    | D ← delete_block_nember (D, N)
16    | update_node_G (N)
17    | while N ←  get_next_node_L () < > 0 do
18    | | b ←  get_number_blocks (N, D)
19    | | update_node_L (N,b)
20    | end
21    | sort_L ()
22   end
```

6.2.2 Performance Analysis of LNBPP

We tested the performance of the proposed LNBPP algorithm using Terasort Benchmark. We only describe the performance of the dual MapReduce since the performance of the single MapReduce is similar. The testing was conducted on the CloudLab [98] which is a cloud

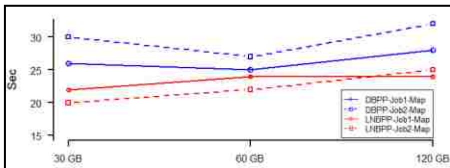
computing platform for research. Our test utilized one master node which manages HDFS, one RM node, and ten slave nodes on the CloudLab. Each node is equipped with 6 GB memory, 2 Xeon E5-2650v2 processors (8 cores each, 2.6 GHz) and 1 TB hard drive. Hadoop 2.7.1 is installed on a Linux machine running Ubuntu 14. Three different data sets have been created by Teragen, i.e., two 30 GB data, two 60 GB data, and two 120 GB data. The data sizes were chosen following the research on Hadoop testing by Intel [101]. For the testing, two jobs with the same input data size were executed, and their results were compared. This is in contrast with the most the previous research where only one job was tested.

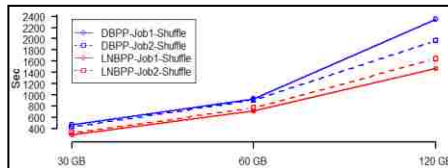
We have measured the execution time of each stage for Map, Shuffle, Merge, Reduce, and the total execution time of MapReduce. The tests were repeated ten times to get the average values. Table 6-1 and Figure 6-3 show the results of the performance testing with default policy

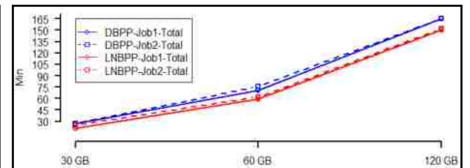
Table 6-1. Terasort with DBPP and LNBPP

	Map (Sec)	Shuffle (Sec)	Total (Min)		Map (Sec)	Shuffle (Sec)	Total (Min)		Map (Sec)	Shuffle (Sec)	Total (Min)
Default-Job1-30G	26	463	27	Default-Job1-60G	25	929	71	Default-Job1-120G	28	2345	165
Default-Job2-30G	30	431	27	Default-Job2-60G	27	906	76	Default-Job2-120G	32	1968	165
LNBPP-Job1-30G	22	292	21	LNBPP -Job1-60G	24	721	59	LNBPP -Job1-120G	24	1475	150
LNBPP-Job2-30G	20	328	25	LNBPP -Job2-60G	22	771	62	LNBPP -Job2-120G	25	1647	152
Improvement (%)	25%	31%	15%	Improvement (%)	12%	19%	18%	Improvement (%)	18%	28%	8%

(a) (b) (c)







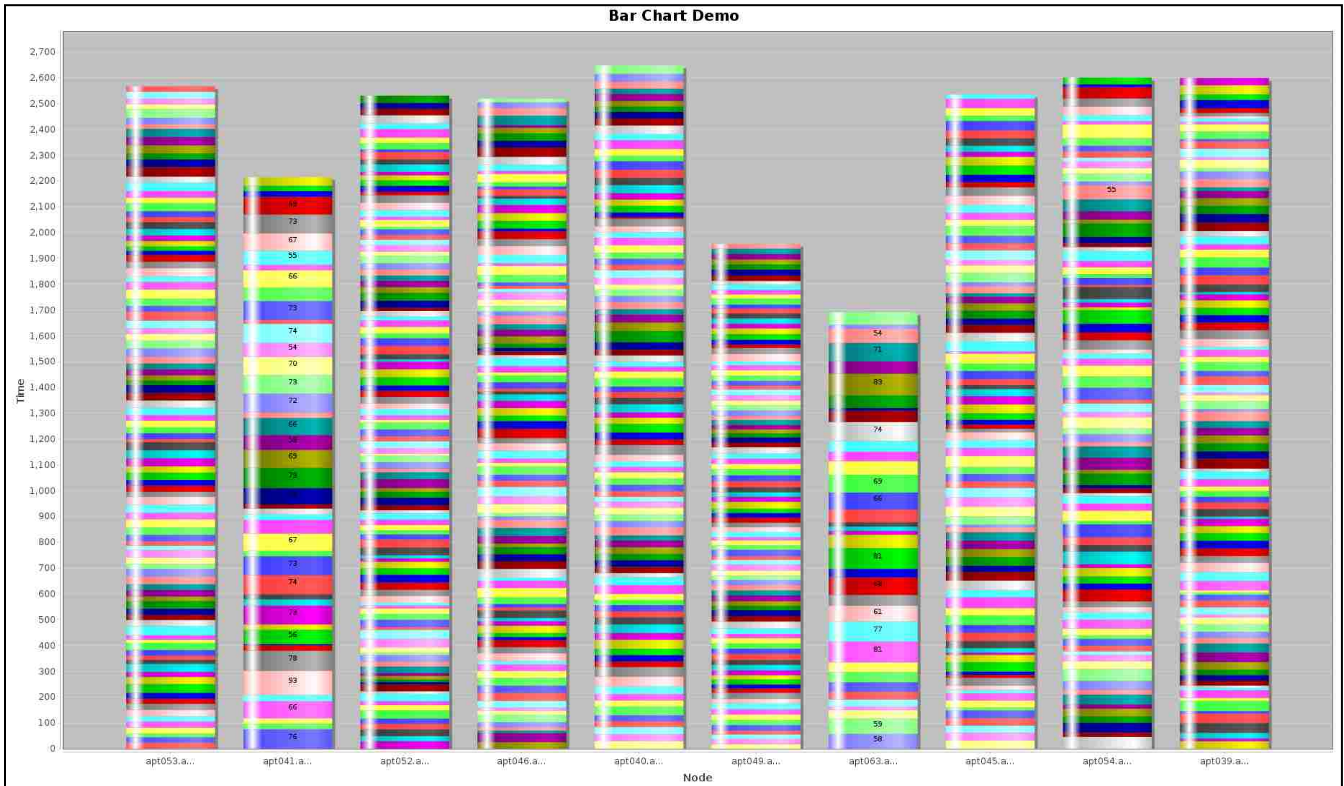
(a) (b) (c)

Figure 6-3 (a) Map Performance, (b) Shuffle Performance, (c) Total Performance

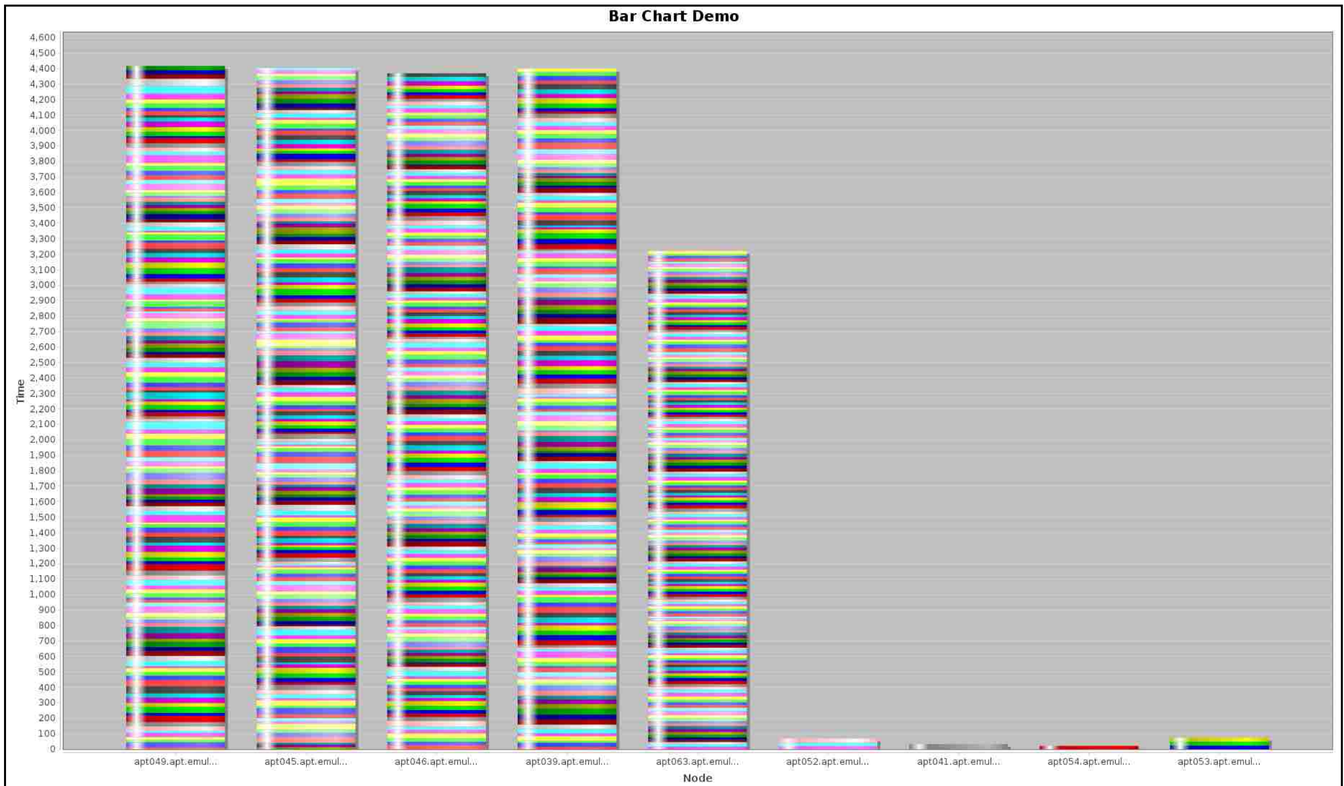
and LNBPP. The "Default-Job1-30G" means "Job1 executing the Default policy with 30 GB data", and the "LNBPP-Job1-30G" means "Job1 executing LNBPP with 30 GB data". In the results, reduce and merge show a similar performance in both cases. However, the map and shuffle show notable differences.

The map performance under LNBPP is faster and more stable than the one under default policy for all sizes of input data. To analyze the performance differences, We have traced all the processes with activated containers in the slave nodes and visualized them in Figure 6-4. Each colored rectangle indicates a container's processing time with one block in Map stage. It shows the differences in container assignments between the default policy where the containers are equally distributed on all slave nodes (Figure 6-4 (a)), and LNBPP where containers are limited to 5 nodes (Figure 6-4 (b)).

After Map stage, the mapper's output has to move to reducer in order to process the job. According to Terasort Benchmark, this Shuffle stage takes more time than the Map stage, therefore decreasing the amount of data transfer in Shuffle stage is the key for improving the overall MapReduce performance. LNBPP on HDFS focuses on Shuffle stage in order to improve the performance of MapReduce. By aggregating the containers in limited nodes (e.g., the left 5 nodes in Figure 6-4 (b)), the amount of data transfer is greatly reduced. In default policy, each node has only roughly 1/10 of data needed for the reduce, so the remaining 9/10 data must be transported. But in LNBPP, each node within the limited nodes group has roughly 1/5 of data, so only 4/5 of data need to be transported. For example, the first slave node apt049 in Figure 6-4 (b) will have a container that becomes a reducer by the LNBPP after finishing Map stage. As a result, apt049 already has roughly 1/5 of five mapper's output needed to process reduce. Therefore, LNBPP can reduce the shuffle time as shown in Figure 6-1.



(a) DBPP



(b) LNBPP

Figure 6-4. An example of mappers' processing times (ms)

In the test, under LNBPP, the mapper's task is assigned to only five nodes, which creates an illusion that it will take more time to finish the Map (T_M). In fact, the histogram shows 2.7 seconds in DBPP and 4.4 seconds in LNBPP. However, T_M is not the sum of all execution times of mapper's tasks. The containers in each node work in parallel using multiple virtual cores. As a result, T_M is determined as the maximum execution time among all the nodes, as we have explained in Section 3. This effect is visualized in Figure 6-4. In case of LNBPP (Figure 6-4 (b)), all rectangle sizes are nearly equal. Any group of 8 containers running on 8 cores independently will finish in almost equal amount of time. However, in case of the DBPP (Figure 6-4 (a)), some nodes, such as apt 041 and apt 063 (2nd and 7th bar in the graph) contain larger rectangles indicating they are taking longer time to finish due to RLM. When they are grouped in 8 containers on 8 cores, the time to complete is the maximum time among those 8 containers. Even though some containers finish early, they cannot start another container in current Hadoop system.

Although a speculative execution is possible, it is only used to take over the work of a crashed node, which is detected by a loss of heartbeat. As a result, the completion time in DBPP is generally larger than in the case of LNBPP. Furthermore, the selection of those slower nodes vary on each execution under DBPP, making the performance fluctuate. The resulting processing times are tabulated in Table 6-1, showing a superior performance of LNBPP. Table 6-1 is the numeric version of Figure 6-3. We observe that the performance is better with LNBPP in all 3 test cases with different data sizes. For Map, the improvement ranged from 12 % to 25%, and for Shuffle it ranged from 19% to 31%. This number is diminished when comparing the total execution time since there are 6 other stages of MapReduce that we have not worked on in this

research. As a result, the total time is improved by 8% to 18%, which is still a significant improvement over DBPP.

6.3 Key-based Deep Data Locality

Key-based DDL is another method to apply DDL into Hadoop. Key-based DDL can handle fine-grain inside block by managing the inside of block unlike block placement researches. Block placement reseraches manage and re-arrange after uploading source data in HDFS as a block, but, key-based DDL manage it before uploading the data in HDFS using big-ETL (Extract, Transfer, Load). The method can reduce data transfer time and merge in MapReduce by useing big-ETL and re-arrange inside of blocks.

Figure 6-5 shows the detail of MapReduce with key-based DDL with comparing default MapReduce. In a typical MapReduce process, after finishing the Map stage, slave nodes send the

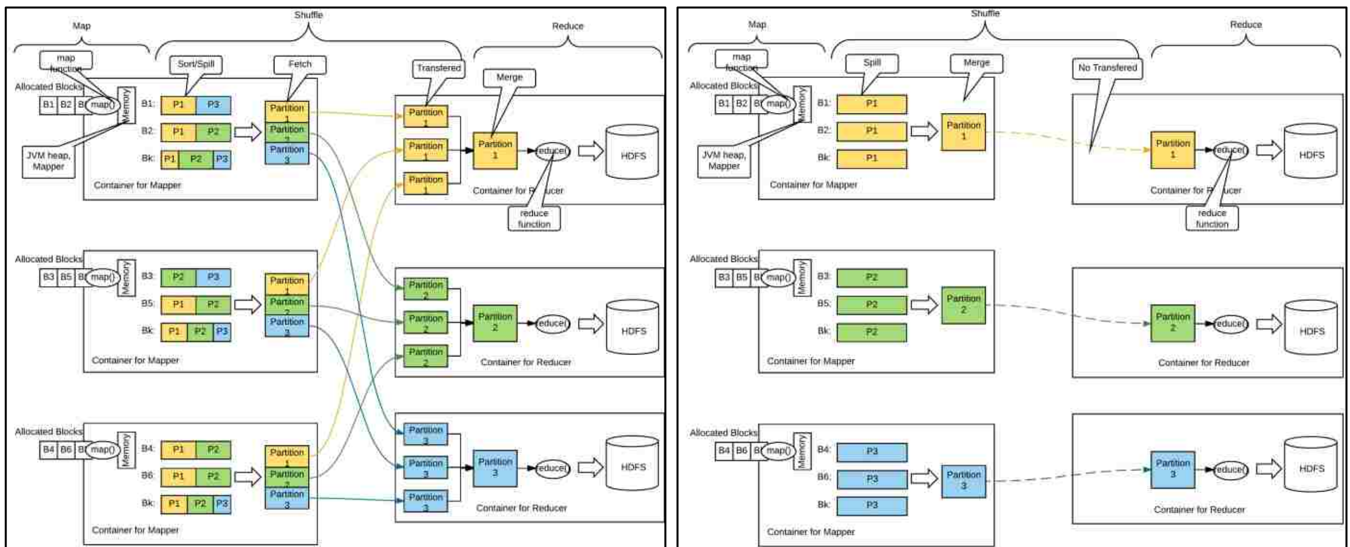


Figure 6-5 (a) Default M/R VS. (b) M/R with Key-Based DDL

partitions that contains pairs of key and value to an appropriate reduce node based on the key as shown in Figure 6-5 (a). There are several keys in default blocks, after finishing map stage, the keys are sorted and fetched on the shuffle stage. The fetched keys are called partitions to transfer to reducer node. The transfred partitions are merged in reducer node to process batch job by reducer function.

The MapReduce with key-based DDL eliminates some stages on shuffle stages and reduce stages to reduce processing time in MapReduce like Figure 6-5 (b). The sort stage and fetch stage on shuffle stage are eliminated by using the same key on the map stage. Also, the merge stage on reduce stage moves to shuffle stage. Normally, merge stage in nodes for the map stage is more efficient than merge stage in nodes for the reduce stage because a number of nodes the for mapper are greater or equal than a number of nodes for the reducer.

Additionally, the amount of transferred partitions can be reduced by selecting node, which has most of key after map stage, as a reducer. In other words, after finishing map stage, YARN changes the node's role, which has most of key, as a mapper to a reducer because both a mapper and a reducer are the container which is daemon on Hadoop. That can minimize network congestion among nodes in MapReduce and increase overall throughput of the system by appling deep data locality.

Basic concept of the key-based DDL is that the nodes which work for MapReduce handle

Figure 6-6 Traditional ETL

the same key to process map stages. There is required pre-step on HDFS before MapReduce to gather the same key in the node. The pre-step can be done by Big-ETL or Hadoop-ETL such as InfoSphere DataStage [106], Data Integrator [107], PowerCenter [108] and so on. In the next section, We will explain about the DDL-Aware ETL for applying key-based DDL into Hadoop.

6.3.1 DDL-Aware ETL

The difference between traditional ETL (Extract, Transfer, and Load) and big-ETL is the procedure of data load on HDFS. Traditional ETL extracts data from source and then the data is transformed to structured data to load in Relational Data Based Management System (RDMS) like Figure 6-6, but is different procedure on big-ETL. Big-ETL use the HDFS instead of RDMS to store the data and to trasfer the data like Figure 6-7 except dash line. Big-ETL load the data before transforming the data into HDFS, which is used for transforming the data, because of the

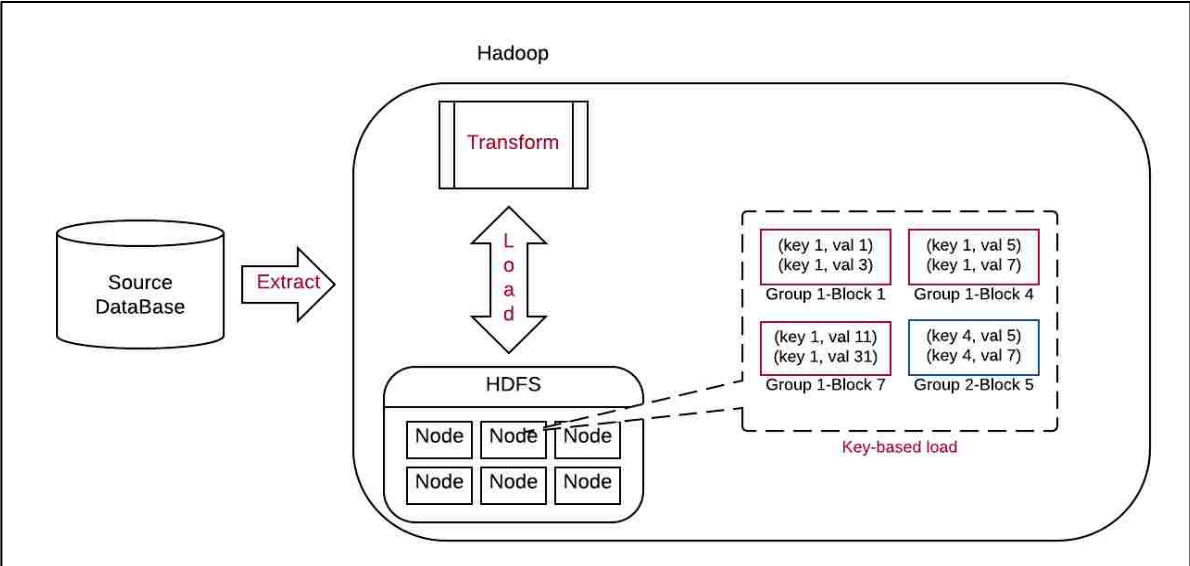
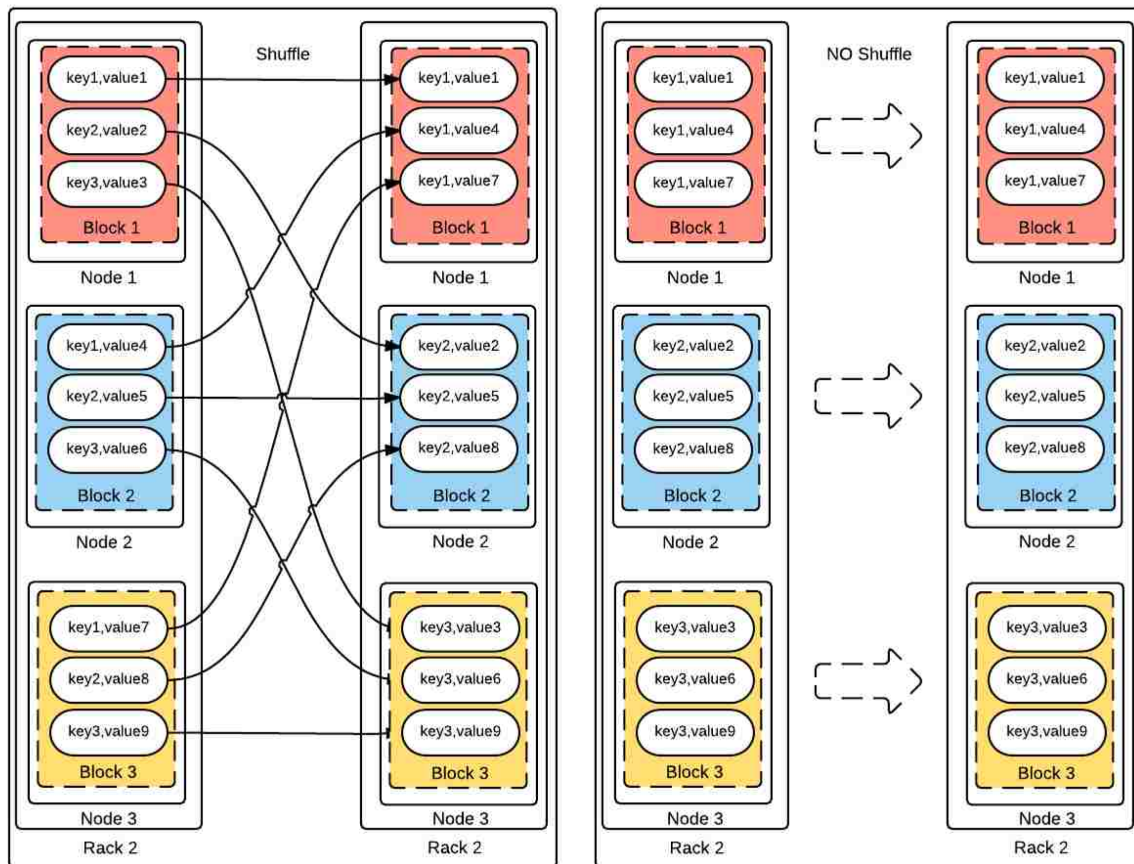


Figure 6-7 DDL-Aware ETL using Big-ETL

volume of data. We used the big-ETL to make a key-based DDL. The big-ETL for key-based DDL is called DDL-Aware ETL. DDL-Aware ETL transform source data into key-based data and making grouping blocks using replicas in HDFS. So, each node load the blocks which have same keys in groups. If the nodes have the block with different key, the block should be in other groups like this Figure 6-7 with dashlines.

Eventually, the purpose of the DDL-Aware ETL is for pre-shuffle in DDL like Figure 6-8 (b). The MapReduce with traditional ETL or big-ETL have no choice but to perform shuffle stage like Figure 6-8 (a). It causes the network traffic and congestion among the nodes unlike



(a) Default block (b) Pre-partitioned block by DDL-Aware ETL

Figure 6-8. Shuffle stage

Figure6-8 (b). Key based DDL uses pre-partition, which is uploaded in HDFS as an input data by DDL-Aware ETL, to perform the shuffle stage like Figure 6-8 (b). In a typical MapReduce process, after finishing the map stage, slave nodes send the partition that contains pairs of key and value to an appropriate reduce node based on the key as shown in Figure 6-8 (a). So, there are an overhead of data transfer in shuffle stage among node. On the other hand, in Figure 6-8 (b), there is no data transfer in shuffle stage after map stages. Figure 6-8 (b) is ideal case.

Key-based DDL improve data locality not only in the map stage but also in the shuffle stage and reducer stage. The data locality in entire stages of MapReduce is very important to increase the performance of Hadoop. In the next section, the paper shows the performance test of key-based DDL.

6.3.2 Performance Analysis of Key-based DDL

Block-based DDL in section 6.2 tests the performance of block-based DDL using Cloud Lab, but We installed Hadoop cluster on physical machines to test of key-based DDL at the Security Lab in the University of Nevada Las Vegas, because the cloud system has various variables, which cause the effect in the performance test of MapReduce.

The cluster is organized 5 machines, one for master node and 4 for data nodes with installed Hadoop 2.6. Each machine has Quad-core Intel Pentium processor, 32GB eMMC disk and 8GB 1333MHz DDR3 memory (Appendix D). The test use 1 giga bytes using default MapReduce, MapReduce with block-based DDL, MapReduce with key-based DDL and MapReduce with key and block-based DDL. There are various methods and combination for Hadoop configure with core-default.xml, hdfs-default.xml and mapred-default.xml. In this test, most of Hadoop configure set as a default except

mapreduce.job.reduce.slowstart.completedmaps = 1 and mapreduce.map.sort.spill.percent = 0.1 in mapred-default.xml.

The Figure 6-9 shows the result of the performance test in map stage. All of the test show the MapReduce with DDL is faster than default MapReduce because there are no Rack Local in map stage by DDL. The processing time of map stage is shorter when using key-based DDL instead of block-based DDL. The reason is why key-DDL already did the part of the map stage's job in an early stage on DDL-Aware ETL like key paring. As the result of the test, map stage is improved 14.6% more than default by key-based DDL and 12.5% more than block-based DDL by key-based DDL.

The locality effect is clearly heard on shuffle stage because network speed is slower than any other hardware devices on Hadoop cluster like disk and memory in Table 5-1. Fig 6-10 shows the result of the performance test in shuffle stage. The DDL tries to minimize the network

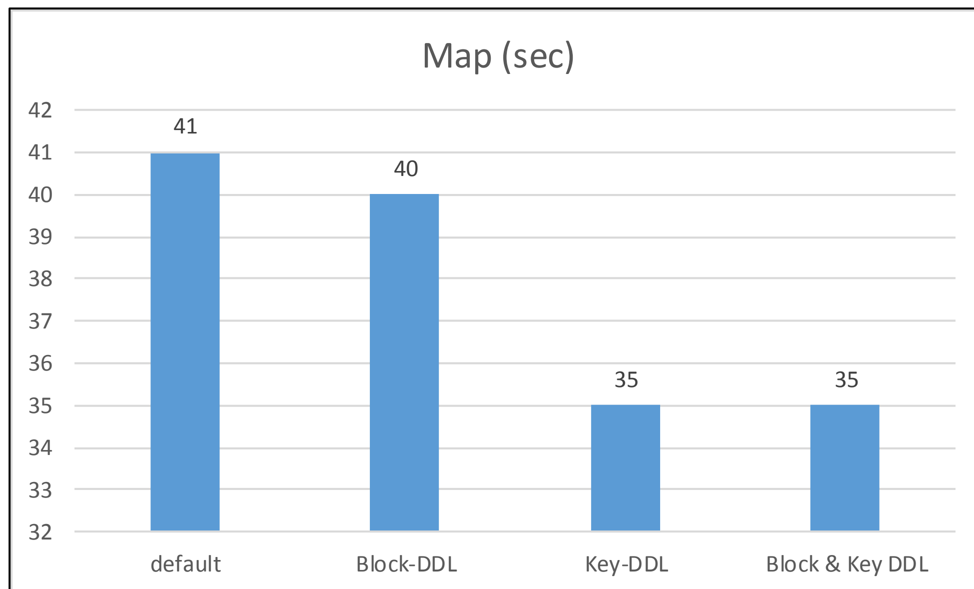


Figure 6-9. Performance test of Map

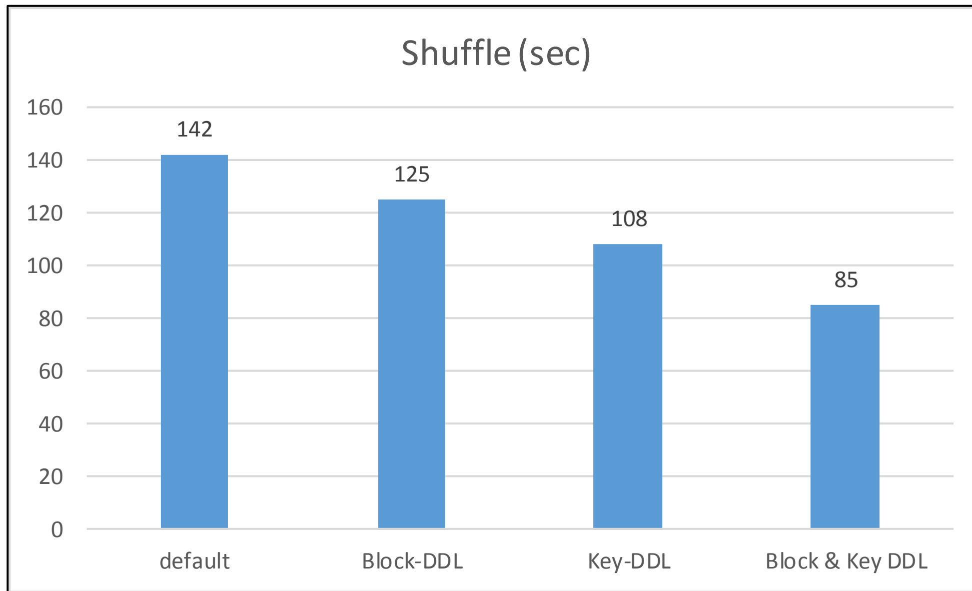


Figure 6-10 Performance test of Shuffle

congestion by reducing data transferring among nodes using data locality. Especially, key-based DDL handles the partition unit in comparison with block-based DDL managed the block unit. In other words, controlling smaller units make more efficient data locality in MapReduce.

According to the result of the test in shuffle stage, the performance of MapReduce is improved upto 23.9% by key-based DDL. The MapReduce with key-based DDL is 13.6% faster than block-based DDL because key-based DDL removing the sort stage and fetch stage in shuffle stage. Additionally, key-based DDL and block-based DDL can be used together to perform MapReduce. The block & key-based DDL is the fastest to perform MapReduce. It improves the performance 34.4% compared with the default MapReduce and 27.2% more than block-based DDL, 16.1% more faster than key-based DDL. The block & key-based DDL has advantages of both block-based DDL, which most of block will be the reducer among the mapper, and key-based DDL, which set each node to have the same key.

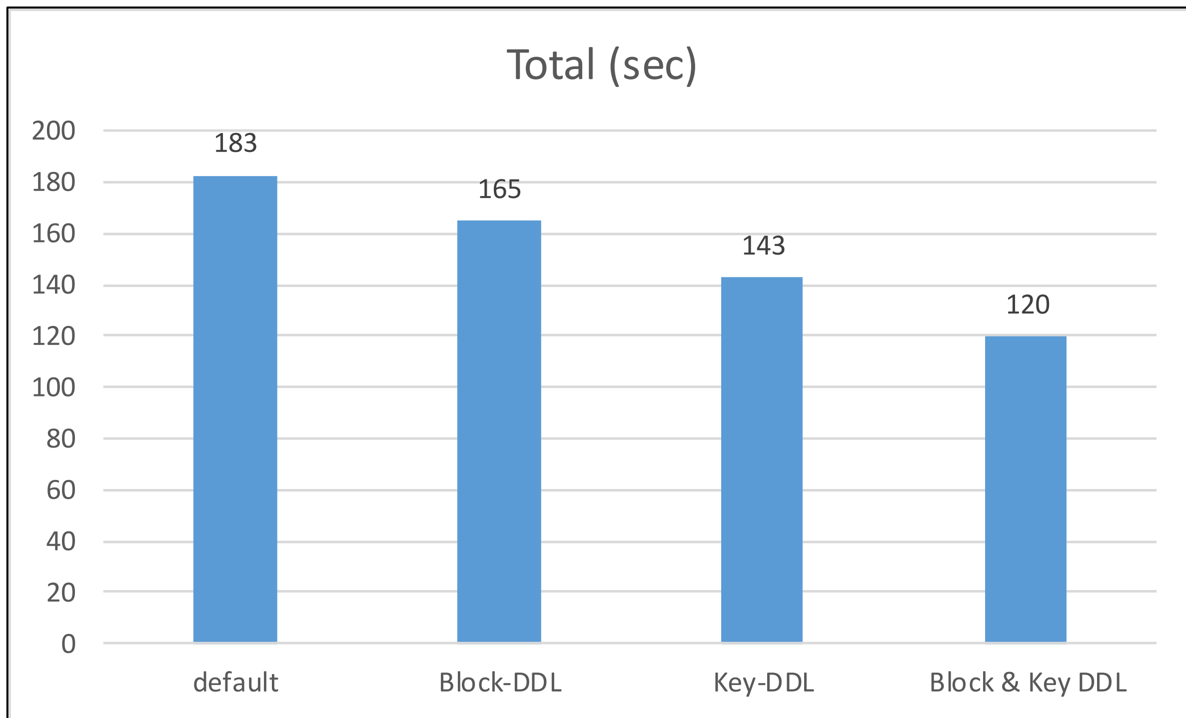


Figure 6-11. Performance test of MapReduce

Figure 6-11 shows the total performance time of MapReduce with the four methods. MapReduce with key-based DDL is 21.9% more faster than default MapReduce and 13.3% more quicker than MapReduce with block-based DDL. Also, when the block-based DDL and key-based DDL are used together for performing MapReduce. Block & Key DDL is 34.4% faster

Table 6-2. Performance improvement by Key-based DDL

	Default	Block-based
Map	14.6%	12.5%
Shuffle	23.9%	13.6%
Total	21.9%	13.3%

Table 6-3. Performance improvement by Block & Key DDL

	Default	Block-based	Key-based
Shuffle	40.1%	32%	21.3%
Total	34.4%	27.2%	16.1%

Table 6-4. Comparison between DDL-aware ETL and Big-ETL

Test cases	Data Size (KB)	Big-ETL (ms)	DDL- aware ETL (ms)	Overhead (%)
1	57,514	1,565	1,677	7.16%
2	575,138	11,939	15,086	26.34%
3	5,751,369	127,543	163,249	27.99%

than default MapReduce. Table 6-2 and Table 6-3 shows the performance improvement by key-based DDL and block & key-based DDL.

Though key-based DDL has some limitations, it can tremendously increase the performance of MapReduce. In the next section, the paper explains about the limitations of key-based DDL.

6.3.3 Limitations of Key-based DDL

Key-based DDL is required an extra stage for setting the same key in the nodes, unlike block-based DDL. The extra stage can cause a delay before procceing MapReduce. We measured the cost to prepare the key-based DDL on HDFS by comparing between DDL-aware ETL and big-ETL. Table 6-4 the result of the measurement. According to the Table 6-4, DDL-aware ETL takes 35 seconds longer than big-ETL to transform 5 GB data. The gap could be resendable because of the increasing performance of Hadoop system by key-based DDL. There is required only extra 5 seconds for 1 GB data with key-based DDL MapReduce to save 40 seconds from default MapReduce. Additionally, there is a room to reduce the gap of the cost by studying ETL.

Even though, there are the limitations, the key-based DDL can reduce the Rack-Local and the Off-Rack in map stage and shuffle stage. When the number of reducers is greater or

equal to the number of mappers, the data transferring among nodes in shuffle stage is removed and Rack-Local and Off-Rack in reduce stage is also removed. The effect of key-based DDL brings enormous effects for real-time analysis by increaing the speed of Hadoop System.

We made the simulator to simulate the data locality researches. We will introduces the MapReduce simulator with various data locality methods in the Appendix A.

CHAPTER 7. CONCLUSIONS AND FUTURE WORK

MapReduce is the fundamental processing paradigm in Hadoop and a lot of research have been conducted to improve Hadoop system performance. We have analyzed previous research comprehensively such as scheduling, data placement, networking, partition/key, and framework, and categorized them in tables and graphs. For example, our analysis shows how the number of containers (mapper and reducer) and data locations affect the performance of MapReduce. Among them, data locality is the primary research direction to improve the performance.

We have surveyed the current research efforts in improving the performance of MapReduce and characterized the processing times in each stage. We made an advanced computational model and simulation system to predict the performance of Hadoop under variety of conditions. A Hadoop operator can test various combinations of data locality to get optimal performance with the characteristics of their data and nodes on the simulator.

Based on the model and the experimental data, we have identified the source of inefficiency during map and shuffle processes. The inefficiency can be mitigated by pre-arranging the data blocks and key-value pairs to the final location rather than the intermediate location in order to reduce the data transfer in later stages. This led to the development of a new concept of Deep Data Locality (DDL) to fully utilize data locality in every stages of MapReduce. There are two methods within DDL, block-based DDL, and key-based DDL.

Block-based DDL removes RLM by placing all the blocks in the destination node in advance, which optimizes the map performance. It also reduces the data transfer during shuffle significantly by selecting the reducer node with the most blocks. The simulation results showed

that block-based DDL outperforms the conventional method by up to 25% in map and 31% in shuffle. The block-based DDL can be used with other methods, which can increase the performance of Hadoop system, especially through an advanced ETL process. Key-based DDL uses the key inside of the block and predict the partitions to remove the network traffic in the shuffle stage. Key-based DDL is required extra stage, which can casue overhead, as a DDL-Aware ETL, but, it can reduce or remove some stages in map stage, shuffle stage and reduce stage. The key-based DDL can dramatically improve MapReduce in some specific case, for example, when the keys in a block are simple or predictable. The MapReduce with key-based DDL is 21.9% faster than default MapReduce and the key-based DDL combined with block-based DDL is 34.4% faster. We made a computational model for key-based DDL to consider the data in the ETL stage.

DDL has a great potential to improve Hadoop performance, but it still need fuether development. Currently, the key-based DDL can be only applied to simple key distributed blocks. The computational model needs to be extended for DDL-Aware ETL to consider any type of data. This research should be extended to include other stages of MapReduce. For example, we have not worked on the merge stage and need to apply the DDL concept to it. Merge stage occupies the largest part in the MapReduce process and any performance improvement in merge stage will have a significant impact in the overall MapReduce performance.

APPENDIX A

Data Locality Simulator

The performance of Hadoop to process MapReduce is affected by a various variable such as number of nodes for mapper, number of nodes for reducer, number of daemons per node, number of blocks, number of keys, distributed keys in block, size of data, complexity of MapReduce program and so on. The simulator is applied the variables to measure the performance of the DDL. Figure A-1 shows the DDL simulator.

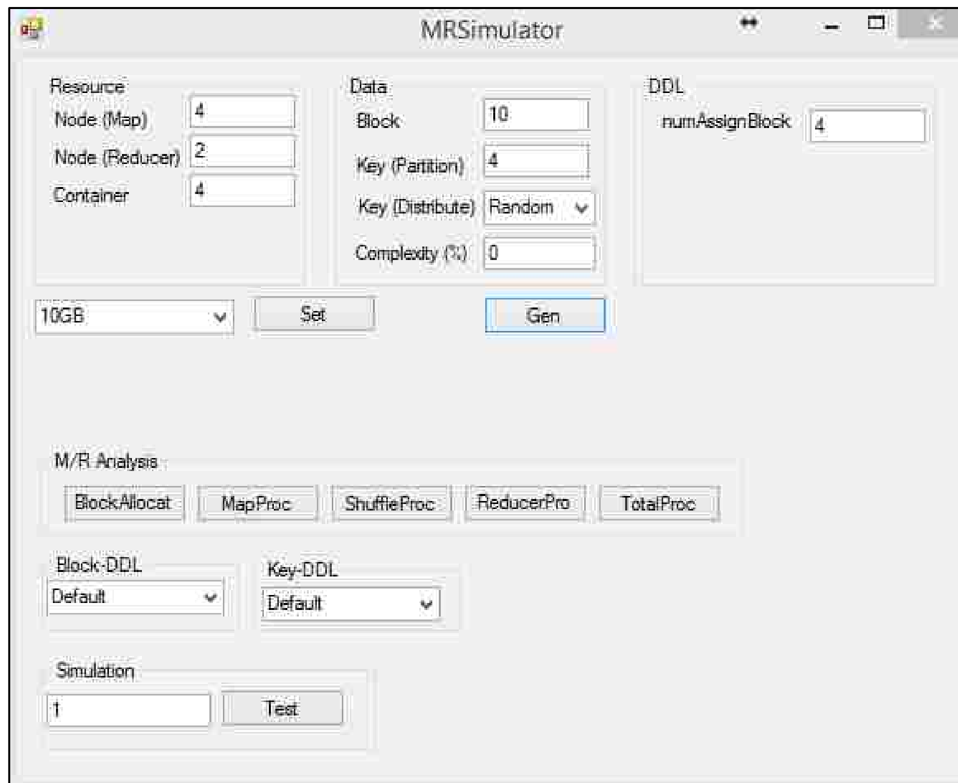


Figure A-1. DDL Simulator

There are two sections to set experiment environments. One is “Resource” section which set the physical machines and daemons in each node. In “Resource” section, the number of nodes for mapper and reducer and number of a container can be managed. By the section, the simulator can select the number of mapper and reducer like Hadoop configuration and MapReduce programming set the number of container for MapReduce processing. The other one is “Data” section for a characteristic of a block. The section can manipulate the input data such as a size of data, the number of blocks, the number of keys for partitions, individual of distributed key in a block, and data complexity. The “DDL” section is for block-based DDL which allocate blocks in nodes. After clicking the “Gen” button, all the physical machines and data is set for

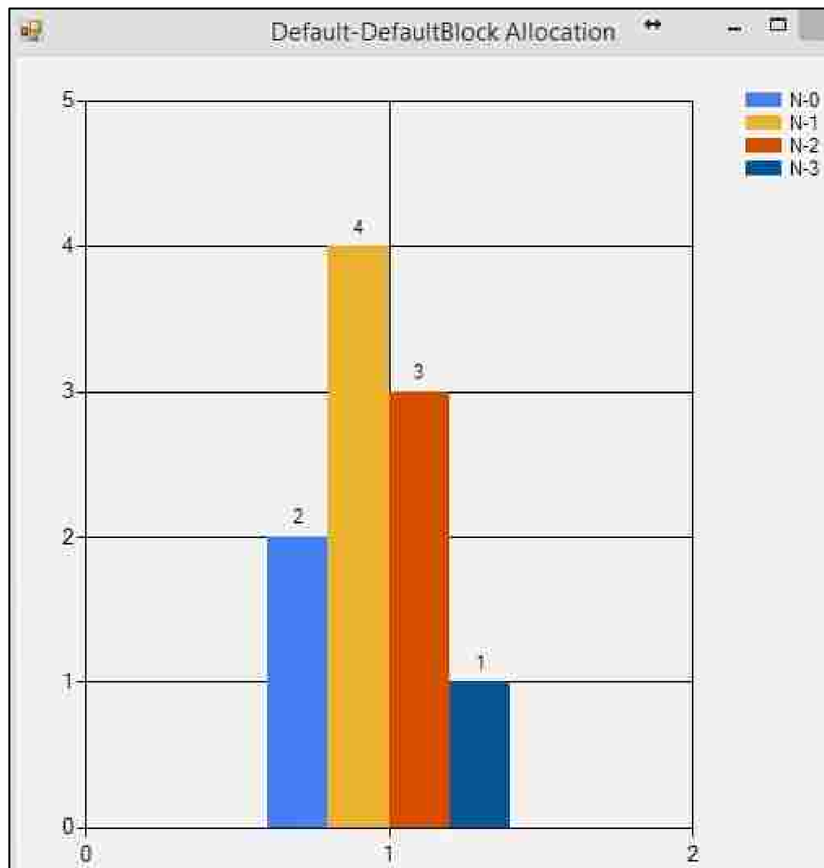


Figure A-2. Block Allocations in the Simulator

MapReduce.

The “M/R Analysis section” is for analyzing the processing time in each stage of MapReduce and allocated blocks in nodes. Figure A-2 shows the example of allocated in nodes when user click the “BlockAllocat” button in the simulator. Each color in Figure A-2 represents node and the number on the graph denotes the number of block in the node.

“MapProc”, “ShuffleProc” and ReducerPro” buttons in the simulator show each process time of MapReduce. Figure A-3 shows the example of analysis in the map stage when a user clicks the “MapProc” button. Each node uses a different number of the container for processing map stage. The max number of container is set by “Resource” section. “M-N0-C3” in Figure A-3 means that “M” is mapper, “N0” is the id of node and “C” is the id of container. In that case,

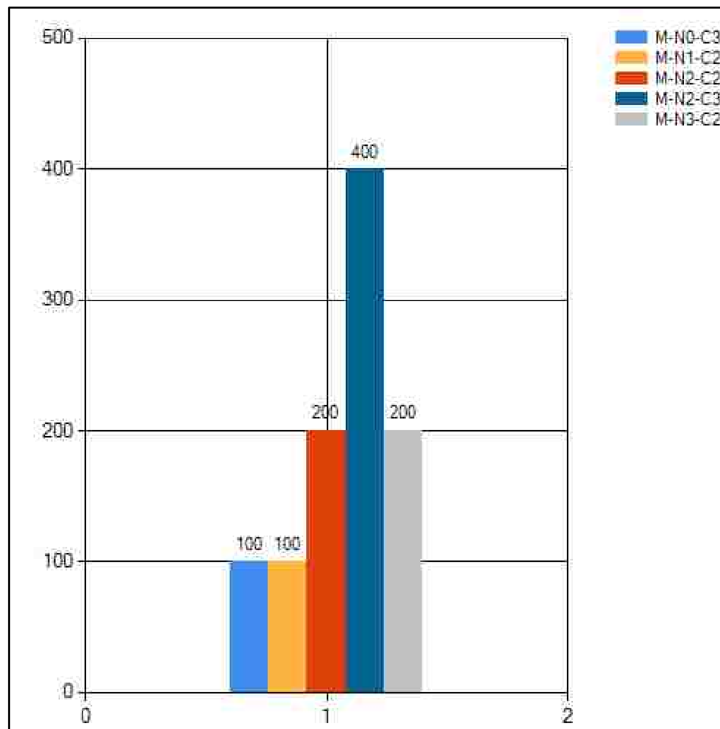
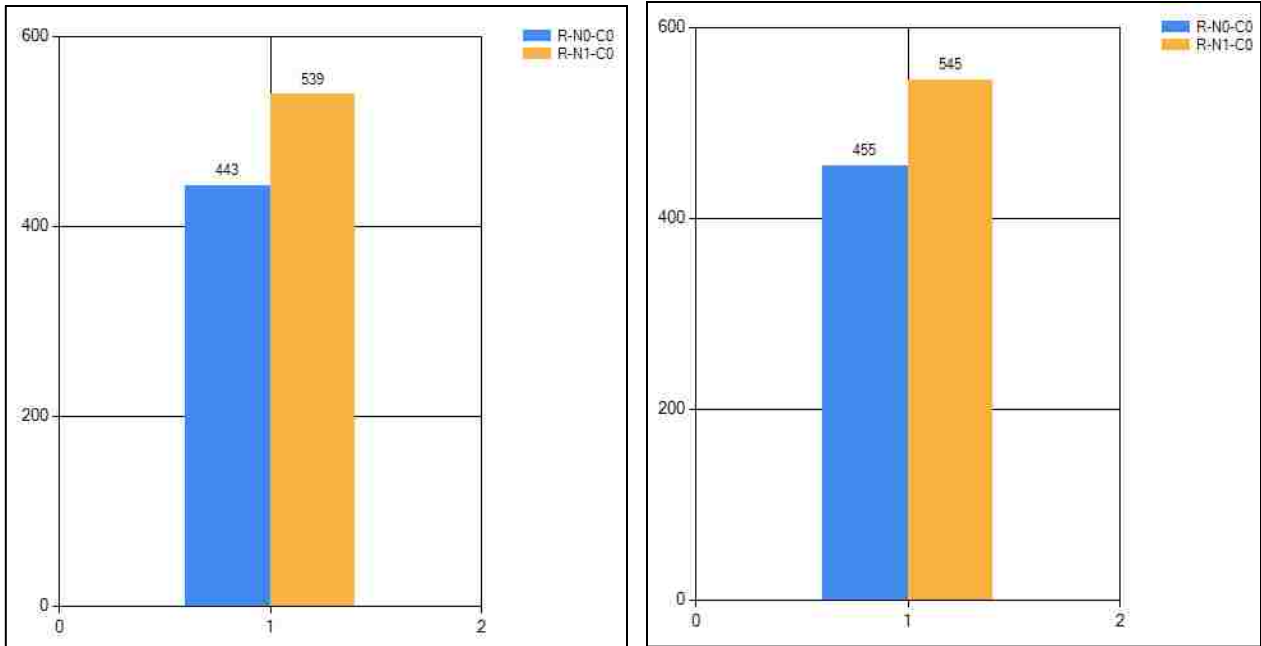


Figure A-3. MapProc



(a) Analysis of the Shuffle (b) Analysis of the Reduce

Figure A-4. ShufflePro and ReducerPro

container “C3” in node-0 performed the map stage as a mapper. The number on top of the graph shows the process time in map stage. Each container takes 100 to process one block, for example, the “M-N2-C3” in Figure A-3 takes 400 processing time with four blocks. Process time of Map in each node is the longest time of container in the node. For example, node-2’s map processing time is 400, even though it has two containers for map stage such as “M-N2-C2” in Figure A-3 and “M-N2-C3” in Figure A-3. The total map processing time is the longest process time of node among the data nodes. The total processing time in Figure A-3 is 400 because the node-2 is the longest processing time of map stage.

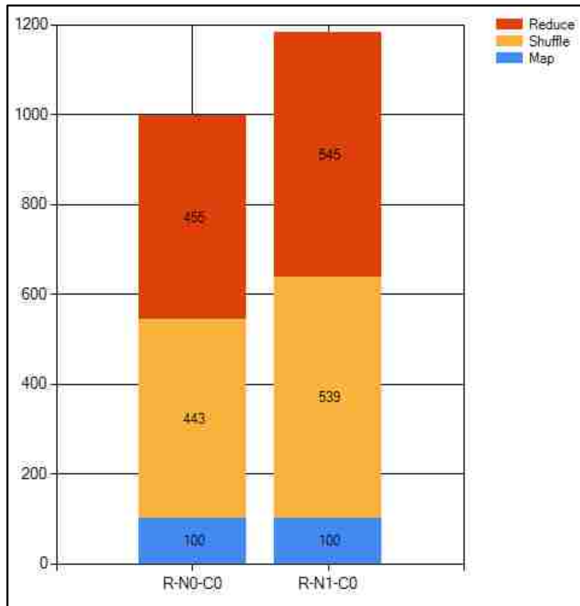
Figure A-4 shows the example of analysis in the shuffle stage and reduce stage when a user clicks the “ShufflePro” and “ReducerPro” button. “R-N0-C0” in the graphs means that “R” is the reducer, “N0” is for node-0 and “C” is for container in node-0. There are two nodes in the

Figure A-4 for processing shuffle stage and reduce stage. Each node has one container. The number in the bar graph means the time unit. The processing time in the Figure A-4 is based on the size of blocks. The reason of the difference of time between Figure A-4 (a) and Figure A-4 (b) is data locality in the simulation. The difference between "R-N1-CO" in Figure A-4 (b) and "R-N1-CO" in Figure A-4 (a) means the saving time by the data local. For example, the 545 time of "R-N1-CO" in Figure A-4 (b) minus the 539 time of "R-N1-CO" in Figure A-4 (a) is 6. The value 6 is the reduced time by the data locality in shuffle stage.

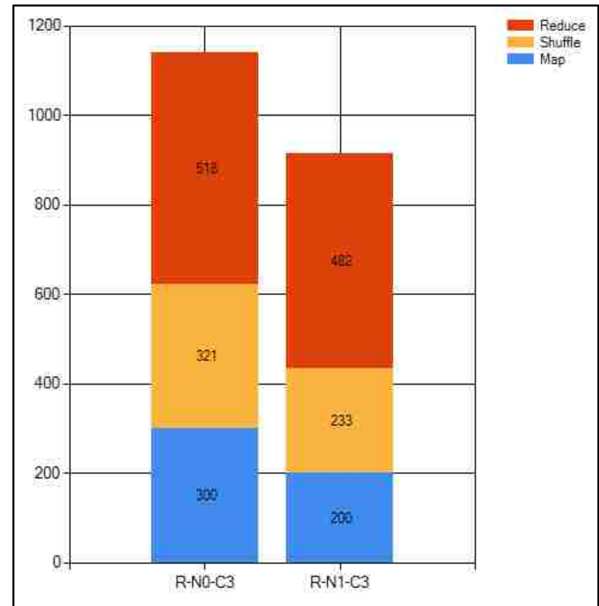
The "TotalProc" button shows the total processing time of MapReduce with map, shuffle and reduce stages. Figure A-5 shows the result of total processing in MapReduce with default, block-based DDL and key-based DDL. The bottom in the graph represents the processing time of map stage. The middle of the bar graph denotes the processing time of shuffle stage. The top of the bar graph represents the processing time of reduce stage. The number in the bar graph means the time unit. The simulator shows the key-based DDL is more efficient than other methods like the previous physical test.

There are two more sections in Figure A-1 to apply the data locality methods on Hadoop. One is "Block-DDL" section to apply MapReduce with block-based DDL. Selecting the methods in the combo box in the section, simulator manipulates the relocation of blocks on HDFS following the rule of block-based DDL. There are two block-based DDL. One is the LNBPP, which set the block location on HDFS before performing the MapReduce. The other one is the Frequent Block Placement Policy (FBPP), which selects the reducer, which has the most block.

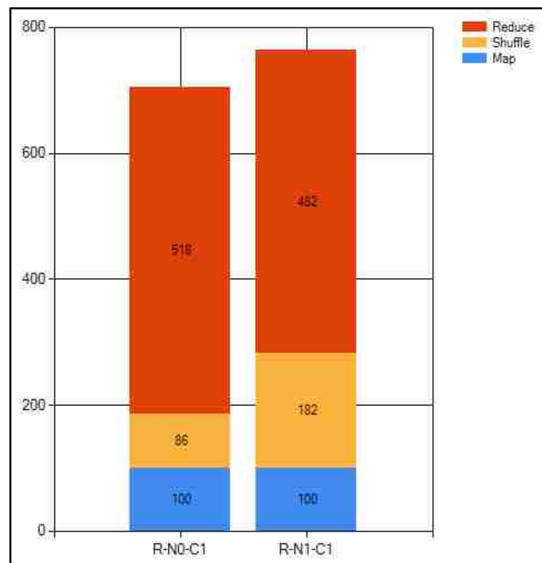
The other section is "Key-DDL" section to apply MapReduce with key-based DDL. Choosing the methods in the combo box in the section, the keys in blocks on HDFS is reallocated following the rule of key-based DDL. There are two key-based DDL methods. One is



Default



Block-Based DDL



Key-Based DDL

Figure A-5. Total processing time of MapReduce with each stage

the keyETL, which gathers keys before uploading HDFS. The other one is Frequent key (FKey), which selects key of the reducer, which has most of the key, as the key collector.

The simulator can make a various combination with block-based DDL and key-based DDL by selecting the methods in the two combo boxes because the two DDL methods can be applied on MapReduce at the same time.

The last section is “Simulation” section which set the number of tests to make an average of processing time. Inputting a number in the box, simulator perform the test as much as the number in the box and then it makes the average of processing time in each stage to display the results.

The simulator makes a various test with DDL methods possible. It lets a result of the performance look easy and can make various combinations with other data locality research. The simulator will make the test of data locality research easy.

APPENDIX B

Sample Test Sheet

We made the log analyzer for Hadoop. We explain the sample of the test result log of the default MapReduce, block-based DDL, key-based DDL, block-based DDL with ETL and key-block-based DDL in this section. The input is 1 giga bytes. The most configuration of Hadoop is the default except "mapreduce.job.reduce.slowstart.completedmaps = 1" and

Table B-1. Default MapReduce

Job Name:	word count		
User Name:	unlv		
Queue:	default		
State:	SUCCEEDED		
Uberized:	FALSE		
Submitted:	Wed Nov 23 18:35:32 PST 2016		
Started:	Wed Nov 23 18:35:41 PST 2016		
Finished:	Wed Nov 23 18:38:33 PST 2016		
Elapsed:	2mins, 51sec		
Diagnostics:			
Average Map Time	41sec		
Average Shuffle Time	2mins, 22sec		
Average Merge Time	0sec		
Average Reduce Time	0sec		
ApplicationMaster			
Attempt Number	**Start Time**	**Node**	**Logs**
1	Wed Nov 23 18:35:35 PST 2016	datanode04:8042	logs
Task Type	**Total**	**Complete**	
Map	16	16	
Reduce	2	2	
Attempt Type	**Failed**	**Killed**	**Successful**
Maps	0	0	16
Reduces	0	0	2
Reducer Node	Node 1	Node 4	

“mapreduce.map.sort.spill.percent = 0.1” in mapred-default.xml. The cluster is organized 5 machines, one for master node and 4 for data nodes with installed Hadoop 2.6. Each machine has Quad-core Intel Pentium processor, 32GB eMMC disk and 8GB 1333MHz DDR3 memory. All the test in this section is used 16 mappers and 2 reducers for processing the word count by the MapReduce programming.

Table B-1 shows the log of default MapReduce. Node-1 and node-4 are used for the reduce stages. The processing time of map stage is 41 sec, and the processing time of shuffle stage is 2 minutes and 22 seconds. Elapsed time is 2 minutes and 51 seconds.

Table B-2. DDL-Aware ETL

Job Name:	word count		
User Name:	unlv		
Queue:	default		
State:	SUCCEEDED		
Uberized:	FALSE		
Submitted:	Wed Nov 23 18:39:40 PST 2016		
Started:	Wed Nov 23 18:39:47 PST 2016		
Finished:	Wed Nov 23 18:42:08 PST 2016		
Elapsed:	2mins, 20sec		
Diagnostics:			
Average Map Time	35sec		
Average Shuffle Time	1mins, 48sec		
Average Merge Time	0sec		
Average Reduce Time	0sec		
ApplicationMaster			
Attempt Number	**Start Time**	**Node**	**Logs**
1	Wed Nov 23 18:39:42 PST 2016	datanode04:8042	logs
Task Type	**Total**	**Complete**	
Map	16	16	
Reduce	2	2	
Attempt Type	**Failed**	**Killed**	**Successful**
Maps	0	0	16
Reduces	0	0	2
Reducer Node	Node 1	Node 2	

Table B-2 shows the log of the applied the DDL-Aware ETL in the MapReduce. Table B-2 is only applied ETL in the Hadoop, is not applied fully DDL like key-based DDL and block-based DDL. Node-1 and node-2 are used for the reduce stages. The processing time of map stage is 35 sec, and the processing time of shuffle stage is 1 minutes and 48 seconds. Elapsed time is 2 minutes and 20 seconds. The processing time of map stage is reduced than the processing time of map stage in default MapReduce in Table B-1, because DDL-Aware ETL can reduce the RLM in the map stage. The processing time of shuffle stage in Table B-2 is also reduced by the DDL-Aware ETL. According to the test, by simply applying the DDL-Aware ETL in the Hadoop, the

Table B-3. Block-based DDL without ETL

Job Name:	word count		
User Name:	unlv		
Queue:	default		
State:	SUCCEEDED		
Uberized:	FALSE		
Submitted:	Wed Nov 23 20:33:01 PST 2016		
Started:	Wed Nov 23 20:33:08 PST 2016		
Finished:	Wed Nov 23 20:35:45 PST 2016		
Elapsed:	2mins, 36sec		
Diagnostics:			
Average Map Time	39sec		
Average Shuffle Time	2mins, 0sec		
Average Merge Time	0sec		
Average Reduce Time	0sec		
ApplicationMaster			
Attempt Number	**Start Time**	**Node**	**Logs**
1	Wed Nov 23 20:33:02 PST 2016	datanode01:8042	logs
Task Type	Total	Complete	
Map	16	16	
Reduce	2	2	
Attempt Type	**Failed**	**Killed**	**Successful**
Maps	0	0	16
Reduces	0	0	2
Reducer Node	Node 1	Node 2	

performance of Hadoop system is increased.

Table B-3 shows the log of MapReduce, which is applied the block-based DDL without the ETL. Table B-3 is only applied block-based ETL in the Hadoop, is not applied any ETL in the HDFS. Node-1 and node-2 are used for the reduce stages. The processing time of map stage is 39 sec, and the processing time of shuffle stage is 2 minutes and 0 seconds. Elapsed time is 2 minutes and 36 seconds. The processing time of map stage is reduced than the processing time of map stage in default MapReduce in Table B-1, because block-based DDL can reduce the RLM in the map stage. The processing time of shuffle stage in Table B-3 is also reduced by the block-based DDL. According to the test, the processing time of the MapReduce with block-based DDL

Table B-4. Block-based DDL with DDL-Aware ETL

Job Name:	word count		
User Name:	unlv		
Queue:	default		
State:	SUCCEEDED		
Uberized:	FALSE		
Submitted:	Wed Nov 23 20:22:41 PST 2016		
Started:	Wed Nov 23 20:22:48 PST 2016		
Finished:	Wed Nov 23 20:25:04 PST 2016		
Elapsed:	2mins, 15sec		
Diagnostics:			
Average Map Time	35sec		
Average Shuffle Time	1mins, 43sec		
Average Merge Time	0sec		
Average Reduce Time	0sec		
ApplicationMaster			
Attempt Number	**Start Time**	**Node**	**Logs**
1	Wed Nov 23 20:22:42 PST 2016	datanode03:8042	logs
Task Type	**Total**	**Complete**	
Map	16	16	
Reduce	2	2	
Attempt Type	**Failed**	**Killed**	**Successful**
Maps	0	0	16
Reduces	0	0	2
Reducer Node	Node 1	Node 2	

in Table B-3 is slower than the processing time of the MapReduce with the DDL-Aware ETL in Table B-2. The part of the map stage is already done by the ETL, because the keys in the blocks are already gathered by ETL in Table B-2. Additionally, the sort stage in Table B-2 is already finished by ETL before processing the MapReduce. Therefore, the test shows that DDL-Aware ETL in Table B-2 seems faster than block-based ETL in Table B-3, but, the DDL-Aware ETL in Table B-2 needs the preprocess for the ETL unlike the block-based ETL. The preporcess for the ETL will make the overhead.

Table B-4 shows the log of MapReduce, which is applied the block-based DDL with DDL-Aware ETL. Table B-4 is applied the ETL in HDFS and block-based in MapReduce. Node-1 and node-2 are used for the reduce stages. The processing time of map stage is 35 sec, and the processing time of shuffle stage is 1 minutes and 43 seconds. Elapsed time is 2 minutes and 15 seconds. The processing time of map stage in Table B-4 is the same as the Table B-2 because of the DDL-Aware ETL. The processing time of shuffle stage in Table B-4 is reduced by block-based ETL and DDL-Aware ETL. The block-based ETL make a synergy with the DDL-Aware ETL, because the two methods work in the different layers.

Table B-5 shows the log of MapReduce, which is applied the key-based DDL. Key-based DDL is already applied the DDL-Aware ETL in HDFS. Node-1 and node-2 are used for the reduce stages. The processing time of map stage is 35 sec, and the processing time of shuffle stage is 1 minutes and 48 seconds. Elapsed time is 2 minutes and 20 seconds. The processing time of map stage in Table B-5 is the same as the Table B-2 because of the DDL-Aware ETL which is required by the key-based DDL. The processing time of shuffle stage in Table B-5 is reduced by the key-based ETL. The key-based DDL is very affected by the number of reducers than other methods. The processing time of shuffle stage in Table B-5 can be reduced by

Table B-5. Key-based DDL

Job Name:	word count		
User Name:	unlv		
Queue:	default		
State:	SUCCEEDED		
Uberized:	FALSE		
Submitted:	Wed Nov 23 18:39:40 PST 2016		
Started:	Wed Nov 23 18:39:47 PST 2016		
Finished:	Wed Nov 23 18:42:08 PST 2016		
Elapsed:	2mins, 20sec		
Diagnostics:			
Average Map Time	35sec		
Average Shuffle Time	1mins, 48sec		
Average Merge Time	0sec		
Average Reduce Time	0sec		
ApplicationMaster			
Attempt Number	**Start Time**	**Node**	**Logs**
1	Wed Nov 23 18:39:42 PST 2016	datanode04:8042	logs
Task Type	**Total**	**Complete**	
Map	16	16	
Reduce	2	2	
Attempt Type	**Failed**	**Killed**	**Successful**
Maps	0	0	16
Reduces	0	0	2
Reducer Node	Node 1	Node 2	

increasing the number of reducers. The key-based DDL can be reduced the dependence of the number of reducers by combining with the block-based ETL.

Table B-6 shows the log of MapReduce, which is applied the key-based DDL and the block -based DDL. Node-2 and node-3 are used for the reduce stages. The processing time of map stage is 35 sec, and the processing time of shuffle stage is 1 minutes and 25 seconds. Elapsed time is 2 minutes and 2 seconds. The processing time of map stage in Table B-6 is the same as the Table B-2 because of the key-based DDL. The processing time of shuffle stage in

Table B-6 is the least time in among the methods, because the key-based DDL and the block-based DDL make the synergy to perform the shuffle stage. The reducer, which has the most of the block, is selected by the block-based DDL, and the reducer is collecting the selected key by the key-based DDL.

Table B-6. Key-Block based DDL

Job Name:	word count		
User Name:	unlv		
Queue:	default		
State:	SUCCEEDED		
Uberized:	FALSE		
Submitted:	Wed Nov 23 19:05:52 PST 2016		
Started:	Wed Nov 23 19:06:02 PST 2016		
Finished:	Wed Nov 23 19:08:13 PST 2016		
Elapsed:	2mins, 2sec		
Diagnostics:			
Average Map Time	35sec		
Average Shuffle Time	1mins, 25sec		
Average Merge Time	0sec		
Average Reduce Time	0sec		
ApplicationMaster			
Attempt Number	**Start Time**	**Node**	**Logs**
1	Wed Nov 23 19:05:56 PST 2016	datanode01:8042	logs
Task Type	**Total**	**Complete**	
Map	16	16	
Reduce	2	2	
Attempt Type	**Failed**	**Killed**	**Successful**
Maps	0	0	16
Reduces	0	0	2
Reducer Node	Node 2	Node 3	

APPENDIX C

Hadoop Configuration

In this appendix, we show configurations of the Hadoop system for the WordCount test. Hadoop has three layers of memory such as YARN, VM, and Java heap to control memory as shown in Figure C-1. YARN manages the memory of the slave node and core. YARN also limits the number of containers in a node based on the memory and core (yarn-site.xml sets the maximum number of core and memory). So, the top layer of memory is yarn.nodemanger.resource.memory in Figure C-1. The memory is set at yarn-site.xml.

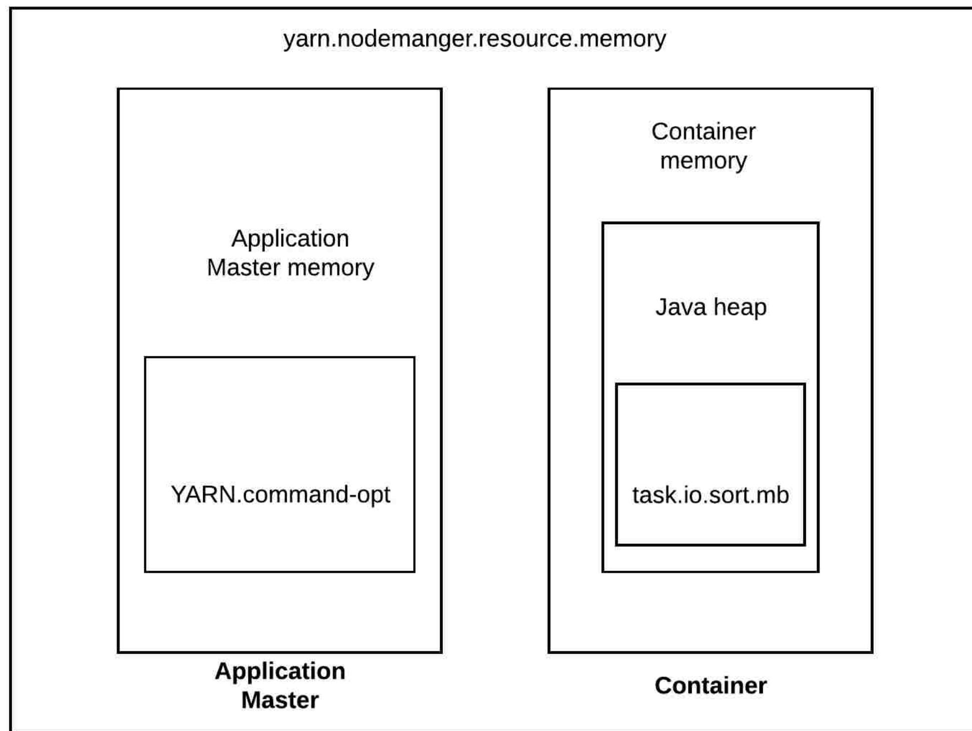


Figure C-1. Memory control in Hadoop.

There are two types of VM when MapReduce starts. Each application has one Application Master (AM). AM manages all containers related to the application. There are two memory options. One is yarn.app.mapreduce.am.resource.mb for the amount of memory the AM needs. The other is yarn.app.mapreduce.am.command-opts for the Java OPTS (OPtional parameTers) for the MapReduce AM processes. The two memory (AM and AM's Java heap) are set by mapred-site.xml. The other VM is a container which is a role of mapper and reducer. There are three types of memory that are set at mapred-site.xml. The task.io.sort.mb is the size of the buffer to use sorting files. The Java heap is for Java OPTS for the task processes. This is set using mapred.child.java.opts in mapred-site.xml. The container memory is the amount of memory that is allocated from the scheduler for mapper/reducer.

The memory setting should be as follows:

Task.io.sort.mb < java heap < Container/AM memory << yarn.nodemanager.resource.memory.

The virtual machines, such as AM and container in slave node, will die if there is not enough memory in the node to allocate container or AM. For example, if the maximum number of containers is four by setting the number of core in configuration and each container can get 2 gigabytes of memory to process the job by setting the size of memory in configuration, then the node should have more then 10 gigabytes because the node needs extra memory for OS and other features – AM, Node Manger (NM) – for managing the node. When the real memory could not support the amount of memory that is required memory size, the VM will die. In the setting, the map, reduce, AM, and Java heap space are allocated an amount of memory. Normally, AM requires 1 gigabyte of memory to manage the containers that process the applications. And,

mapper and reducer require more memory than Java heap space. Java heap space requires enough memory to process a job in a container. There is Java heap space error when the Java heap space is not sufficiently allocated to process the job in the container.

Tables C-1, C-2, and C-3 show the Hadoop configurations for the WordCount test. Mostly, the resource of VM is controlled at yarn-site.xml and mapred-site.xml. Also, the input data is controlled as blocks on HDFS by hdfs-site.xml.

In the WordCount test with 1 gigabyte data, we used one master node and four slave nodes, with 8 gigabytes real memory and four virtue cores each. So, we set the reducer to work simultaneously with a map to process the MapReduce job after finishing a half of map stage using mapreduce.job.reduce.slowstart.completedmaps in mapred-site.xml. One slave node in the slave nodes will work as an AM that will use one virtual core, 1 gigabyte memory (yarn.app.MapReduce.am.resource.mb) for AM's resource, and 768 megabytes of Java heap memory (yarn.app.MapReduce.am.command-opts) using mapred-site.xml. According to the setting, all slave nodes can get maximum two mappers. There are two reducers in the slave nodes. We allocated each mapper and reducer one virtue and 2600 megabytes of memory (mapreduce.map/reduce.memory.mb) using mapred-site.xml. Therefore, we limited the maximum number of virtual machines to four in the slave nodes using yarn-site.xml and mapred-site.xml.

Using the hdfs-site.xml, we set the size of the block at 64 megabytes using dfs.blocksize and made replicas using dfs.replication in hdfs-site.xml.

Table C-1. hdfs-site.xml.

Name	Value
dfs.replication	3
dfs.permissions	false
dfs.blocksize	64m
dfs.namenode.name.dir	/home/unlv/Hadoop_tmp/hdfs/namenode
dfs.datanode.data.dir	/home/unlv/Hadoop_tmp/hdfs/datanode

Table C-2. mapred-site.xml.

Name	Value
mapreduce.[map|reduce].cpu.vcores	1
yarn.app.MapReduce.am.command-opts	-Xmx768m
yarn.app.MapReduce.am.resource.mb	1024
mapred.child.java.opts	-Xmx2560m
mapreduce.task.io.sort.mb	256
mapred.job.shuffle.input.buffer.percent	0.10
mapreduce.map.java.opts	-Xmx2560m
mapreduce.reduce.java.opts	-Xmx2560m
mapreduce.map.memory.mb	2600
mapreduce.reduce.memory.mb	2600
mapreduce.job.reduce.slowstart.completedmaps	0.5
mapreduce.tasktracker.map.tasks.maximum	1
mapreduce.tasktracker.reduce.tasks.maximum	1
mapreduce.framework.name	yarn
mapred.job.shuffle.input.buffer.percent	0.10
MapReduce.jobhistory.address	master:10020
MapReduce.jobhistory.webapp.address	master:19888
MapReduce.jobhistory.max-age-ms	5184000000
MapReduce.jobhistory.intermediate-done-dir	/home/unlv/Hadoop/mr-history/tmp
MapReduce.jobhistory.done-dir	/home/unlv/Hadoop/mr-history/done
MapReduce.map.speculative	false
MapReduce.reduce.speculative	false
mapred.map.tasks.speculative.execution	false
mapred.reduce.tasks.speculative.execution	false

Table C-3. yarn-site.xml

Name	Value
yarn.scheduler.maximum-allocation-vcores	2
yarn.nodemanager.resource.cpu-vcores	2
yarn.nodemanager.resource.memory-mb	7000
yarn.scheduler.maximum-allocation-mb	7000

APPENDIX D

Hardware Implement

We tested the data locality performances with hardware implementation to verify the accuracy of the analytical and simulation models. A small Hadoop cluster has been configured with five machines, one for master node and 4 for slave nodes (Figure D-1). Each machine is equipped with quad-core Intel Pentium processor, 32GB eMMC disk (250 MB/s) and 8GB of 1333MHz DDR3 memory. A Netgear GS108-Tv2 switch (100 Mbps) is used for the network. Using WordCount benchmark with 1 GB of input data, we tested four different methods, i.e., default MR, MR with block-based DDL, MR with key-based DDL, and MR with both block-based and key-based DDL.

We also installed the Ganglia 3.6 to check resource in the Hadoop system [112]. Ganglia store the log at default location (default: "/var/lib/ganglia/rrds"). The log of Ganglia will make full of size in the root directory. There is needed to move default location to other location.

Figure D-1 Hadoop Testbed

REFERENCES

[1] Sungchul, Lee., Juyeon, Jo., Yoohwan, Kim., “Restful Web Service and WebBased Data Visualization for Environmental Monitoring”, International Journal of Software Innovation, 3(1), Jan. 2015. pp. 75-95.

[2] IBM “What is big data?”. Retrieved: Mar. 13, 2015. Online available: http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html

[3] IBM “InfoSphere BigInsight”. Retrieved: Mar. 13, 2015, Online available: http://www-03.ibm.com/software/products/en/infobigi

[4] Oscar D. Lara, Weiqiang Zhuang, and Adarsh Pannu “Big R: Large-scale Analytics on Hadoop using R”, 2014 IEEE International Congress on Big Data, June 27 2014, pp. 570-577

[5] Revolution analytics “RHadoop and MapR” 14, Mar. 2014

[6] Spark “Apache SpartTM “Mar, 13, 2015, Online available: https://spark.apache.org/

[7] Global Futures and Foresight, “The Futures Report 2011,” Online available: www.steria.com/futuresreport

[8] “Apache Hadoop,”.Retrieved: Mar, 13, 2015. Online available: http://hadoop.apache.org

[9] Sungchul Lee, Ju-Yeon Jo and Yoohwan Kim, “Data analysis performance comparison between single-mode and multi-mode,” 25th International Conference on Software Engineering and Data Engineering, SEDE 2016, pp. 47-52.

[10] Hadoop. Retrieved from https://Hadoop.apache.org/docs/r1.2.1

[11] IBM, “What is the Hadoop Distributed File System (HDFS)?” Retrieved from https://www-01.ibm.com/software/data/infosphere/hadoop/hdfs/

[12] M. Zaharia, et al., "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in Proceedings of the 5th European conference on Computer systems, 2010, pp. 265-278.

[13] M. Elteit, H. Lin, and W. Feng, “Enhancing MapReduce via Asynchronous Data Processing,” in Proceedings of IEEE 16 International Conference on Parallel and Distributed Systems (ICPADS), pp. 397-405, December 2010.

[14] K. Kambatla, N. Rapolu, S. Jagannathan, and A. Grama, “Asynchronous Algorithm in MapReduce,” in Proceedings - IEEE International Conference on Cluster Computing (ICCC), pp. 245-254, September 2010.

[15] M. Isard, Vijayan Prabhakaran, J. Currey et al., “Quincy: fair scheduling for distributed computing clusters,” in SOSP’09, 2009, pp. 261–276.

[16] Tseng-Yi Chen, Hsin-Wen Wei, Ming-Feng Wei, Ying-Jie Chen, Tsan-sheng Hsu and Wei-Kuan Shih “LaSA: A locality-aware scheduling algorithm for Hadoop-MapReduce resource assignment,” Collaboration Technologies and Systems (CTS), 2013 International Conference on, 20-24 May 2013, pp. 342 – 346.

[17] Liying Li, Zhuo Tang, Renfa Li and Liu Yang. “New improvement of the Hadoop relevant data locality scheduling algorithm based on LATE,” 2011 International

Conference on Mechatronic Science, Electric Engineering and Computer (MEC), 2011. pp. 1419 – 1422

[18] Zhenhua Guo, Geoffrey Fox and Mo Zhou. “Investigation of Data Locality in MapReduce,” 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012) Year: 2012. Pages: 419 – 426

[19] Benjamin Heintz, Chenyu Wang, Abhishek Chandra and Jon Weissman, “Cross-Phase Optimization in MapReduce,” 2013 IEEE International Conference on Cloud Engineering (IC2E), 2013 pp. 338 – 347

[20] Phuong Nguyen, Tyler Simon, Milton Halem, David Chapman and Quang Le, “A Hybrid Scheduling Algorithm for Data Intensive Workloads in a MapReduce Environment,” 2012 IEEE Fifth International Conference on Utility and Cloud Computing, 2012. pp. 161 – 167

[21] Xiaohong Zhang, Zhiyong Zhong, Shengzhong Feng, Bibo Tu and Jianping Fan, “Improving Data Locality of MapReduce by Scheduling in Homogeneous Computing Environments,” 2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications, 2011. pp. 120 – 126.

[22] Zhuo Tang, Junqing Zhou, Kenli Li and Ruixuan Li, “MTSD: A Task Scheduling Algorithm for MapReduce Base on Deadline Constraints,” 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, 2012. pp. 2012 – 2018.

[23] Xie, J., Yin, S., Ruan, X., Ding, Z., Tian, Y., Majors, J., Manzanares, A., and Qin, X. “Improving MapReduce performance through data placement in heterogeneous Hadoop clusters,”. In Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on (2010), pp. 1–9.

[24] Shadi Ibrahim, Hai Jin, Lu Lu, Bingsheng He, Gabriel Antoniu and Song Wu “Maestro: Replica-Aware Map Scheduling for MapReduce,” 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2012. pp. 435-442.

[25] Lei Lei, Tianyu Wo and Chunming Hu, “CREST: Towards Fast Speculation of Straggler Tasks in MapReduce,” 2011 IEEE 8th International Conference on e-Business Engineering, 2011. pp. 311 – 316.

[26] Cristina L. Abad; Yi Lu; Roy H. Campbell, “DARE: Adaptive Data Replication for Efficient Cluster Scheduling,” 2011 IEEE International Conference on Cluster Computing, 2011, pp. 159 – 168.

[27] Tang, S., Lee, B. S., and He, B. DynamicMR: A Dynamic Slot Allocation Optimization Framework for MapReduce Clusters. Transactions on Cloud Computing, VOL. 2, NO. 3, pp. 333-347. IEEE, 2014.

[28] Elshater, Y. and Martin, P. A Study of Data Locality in YARN, IEEE International Congress on Big Data, pp. 174 – 181. IEEE, 2015.

[29] Nishanth S., Radhikaa B., Ragavendar T. J., Babu, C., and Prabavathy B. CoRadoop++: A Load Balanced Data Colocation in Radoop Distributed File System, Fifth International Conference on Advanced Computing, ICoAC, pp. 100 – 105. IEEE, 2013.

[30] Membrey, P., Chan, K. C.C., Demchenko, Y. A Disk Based Stream Oriented Approach For Storing Big Data, International Conference on Collaboration Technologies and Systems, CTS, pp. 56 – 64. IEEE, 2013.
[31] Hou, X., Ashwin Kumar T K, Thomas, J.P. and Varadharajan, V. Dynamic Workload Balancing for Hadoop MapReduce, IEEE Fourth International Conference on Big Data and Cloud Computing, pp. 56-62. IEEE, 2014.
[32] AMD, Hadoop Performance Tuning Guide. Rev 1.0. Retrieved from http://www.admin-magazine.com/HPC/Vendors/AMD/Whitepaper-Hadoop-Performance-Tuning-Guide. 2012.
[33] Dai, X., and Bensaou, B. A Novel Decentralized Asynchronous Scheduler for Hadoop, Communications QoS, Reliability and Modelling Symposium, Globecom`s 2013, pp. 1470 – 1475. IEEE, 2013.
[34] Zhu, H. and Chen, H. Adaptive Failure Detection via Heartbeat under Hadoop, IEEE Asia -Pacific Services Computing Conference, pp. 231 – 238. IEEE, 2011.
[35] Yan, J., Yang, X., Gu,R., Yuan C. and Huang, Y. Performance Optimization for Short MapReduce Job Execution in Hadoop, Second International Conference on Cloud and Green Computing, pp. 688 – 694. IEEE, 2012.
[36] Qin, P., Dai, B., Huang, B. and Xu, G. Bandwidth-Aware Scheduling With SDN in Hadoop: A New Trend for Big Data, IEEE Systems Journal. Vol. PP, Issue:99 20, pp. 1-8. IEEE, 2015.
[37] Lu, X., Nusrat S. Islam, Md. Wasi-ur-Rahman, M., Jose, J., Subramoni, H., Wang, H, and Panda, D. K. High-Performance Design of Hadoop RPC with RDMA over InfiniBand, 42nd International Conference on Parallel Processing, pp. 641 – 650. IEEE, 2013.
[38] Basak, A., Brnster, I., Ma, X. and Mengshoel, O.J. Accelerating Bayesian network parameter learning using Hadoop and MapReduce, Proceedings of the 1st International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications, BigMine ’12, pp. 101-108. ACM, 2012.
[39] Iwazume, M., Iwase, T., Tanaka, K., et al. Big Data in Memory: Benchimarking In Memory Database Using the Distributed Key-Value Store for Machine to Machine Communication, 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD, pp. 1 – 7. IEEE/ACIS, 2014.
[40] SAP HANA. Retrieved from http://hana.sap.com/abouthana.html.
[41] Apache Spark. Retrieved from http://spark.apache.org/.
[42] Islam, N.S., Wasi-ur-Rahman, M., Lu, X., Shankar, D. and Panda D. K. Performance Characterization and Acceleration of In-Memory File Systems for Hadoop and Spark Applications on HPC Clusters, International Conference on Big Data, IEEE Big Data, pp. 243 – 252. IEEE, 2015.
[43] Uchigaito, H., Miura, S. and Nito, T. A Control Scheme for Eliminating Garbage Collection during High-speed Analysis of Big-graph Data Stored in NAND Flash Memory, IEEE International Symposium on Circuits and Systems (ISCAS), pp. 2557 – 2560. IEEE, 2015.

[44] Mohamed Y. Eltabakh, Yuanyuan Tian, Fatma Ozcan Rainer Gemulla, Aljoscha Krettek and John McPherson. CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop. Journal Proceedings of the VLDB Endowment, Volume 4 Issue 9, June 2011. Pages 575-585.

[45] Asmath Fahad Thaha, Anang Hudaya Muhamad Amin, Subarmaniam Kannan and Nazrul Muhaimin Ahmad,"Data Location Aware Scheduling for Virtual Hadoop Cluster Deployment on Private Cloud Computing Environment," IEEE Communications (APCC), 2016 22nd Asia-Pacific Conference on. 25-27 Aug. 2016, pp. 103 – 109.

[46] Ruiqi Sun, Jie Yang, Zhan Gao and Zhiqiang He. "A virtual machine based task scheduling approach to improving data locality for virtualized Hadoop," Computer and Information Science (ICIS), 2014 IEEE/ACIS 13th International Conference on, 4-6 June 2014. pp. 297 – 302

[47] Park, J., Lee, D., Kim, B., Huh, J., and Maeng, S. "Locality-aware dynamic vm reconfiguration on MapReduce clouds," In Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing (New York, NY, USA, 2012), HPDC '12, ACM, pp. 27–36.

[48] Palanisamy, B., Singh, A., Liu, L., and Jain, B. "Purlieus: locality-aware resource allocation for MapReduce in a cloud," In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (New York, NY, USA, 2011), SC '11, ACM, pp. 58:1–58:11.

[49] Xiaoqiang Ma, Xiaoyi Fan, Jiangchuan Liu and Dan Li. "Dependency-aware Data Locality for MapReduce," IEEE Transactions on Cloud Computing. 2015, Volume: PP, Issue: 99. Pages: 1 – 13.

[50] Package org.apache.hadoop.examples.terasort Retrieved from : https://hadoop.apache.org/docs/current/api/org/apache/hadoop/examples/terasort/package-summary.html

[51] Dai, X., and Bensaou, B. A Novel Decentralized Asynchronous Scheduler for Hadoop, Communications QoS, Reliability and Modelling Symposium, Globecom`s 2013, pp. 1470 – 1475. IEEE, 2013.

[52] Zhu, H. and Chen, H. Adaptive Failure Detection via Heartbeat under Hadoop, IEEE Asia -Pacific Services Computing Conference, pp. 231 – 238. IEEE, 2011.

[53] Yan, J., Yang, X., Gu,R., Yuan C. and Huang, Y. Performance Optimization for Short MapReduce Job Execution in Hadoop, Second International Conference on Cloud and Green Computing, pp. 688 – 694. IEEE, 2012.

[54] Shanjiang Tang, Bu-Sung Lee and Bingsheng He, "DynamicMR: A Dynamic Slot Allocation Optimization Framework for MapReduce Clusters. Transactions on Cloud Computing," VOL. 2, NO. 3, pp. 333-347. IEEE, 2014.

[55] Yongliang Xu and Wentong Cai, "Hadoop Job Scheduling with Dynamic Task Splitting," 2015 International Conference on Cloud Computing Research and Innovation (ICCCRI), 2015, pp. 120 – 129

[56] Supriya Pati and Mayuri A. Mehta, "Job aware scheduling in Hadoop for heterogeneous cluster," Advance Computing Conference (IACC), 2015 IEEE International, 12-13 June 2015. pp. 778-783.

[57] Bing Zhao, Yun Liu and Bo Shen,“The Dynamic Delay Scheduling Algorithm Based on Task Classification,” 2015 First International Conference on Computational Intelligence Theory, Systems and Applications (CCITSA), 2015. pp. 92 – 97

[58] Zhuo Tang, Wen Ma, Kenli Li and Keqin Li, “A Data Skew Oriented Reduce Placement Algorithm Based on Sampling,” IEEE Transactions on Cloud Computing, 2016, Volume: PP, Issue: 99. pp. 1 – 14.

[59] Microsoft Azure. https://azure.microsoft.com

[60] Amazon Web Services https://aws.amazon.com/

[61] Weina Wang and Lei Ying, “Data locality in MapReduce: A network perspective,” Communication, Control, and Computing (Allerton), 2014 52nd Annual Allerton Conference on, 30 Sept.-3 Oct. 2014. pp. 1110-1117.

[62] Huang Suyu, “Data Mining Association Rule Algorithm Based on Hadoop,” 2015 8th International Conference on Intelligent Computation Technology and Automation (ICICTA), 2015, pp. 349 – 352.

[63] Sungchul Lee, Ju-Yeon Jo, Yoohwan Kim, “Performance Improvement of MapReduce Process by Promoting Deep Data Locality,” Data Science and Advanced Analytics (DSAA), 2016 IEEE International Conference on, 2016. pp. 292-301.

[64] Jun Wang, Qiangju Xiao, Jiangling Yin and Pengju Shang, “DRAW: A New Data-gRouping-AWare Data Placement Scheme for Data Intensive Applications With Interest Locality,” IEEE Transactions on Magnetics, 2013, Volume: 49, Issue: 6. pp 2514 – 2520.

[65] Pengju Shang, Qiangju Xiao and Jun Wang, “DRAW: A new Data-gRouping-AWare data placement scheme for data intensive applications with interest locality,” 2012 Digest APMRC, 2012. pp. 1 – 8.

[66] Vrushali Ubarhande, Alina-Madalina Popescu and Horacio González-Vélez “Novel Data-Distribution Technique for Hadoop in Heterogeneous Cloud Environments,” 2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems, 2015. pp. 217 – 224.

[67] Jen Chun Hsu, Ching Hsien Hsu, Shih Chang Chen and Yeh Ching Chung, “Correlation Aware Technique for SQL to NoSQL Transformation,” 2014 7th International Conference on Ubi-Media Computing and Workshops, 2014. pp. 43 – 46.

[68] Hui Jin, Xi Yang, Xian-He Sun and Ioan Raicu , “ADAPT: Availability-Aware MapReduce Data Placement for Non-dedicated Distributed Computing,”2012 IEEE 32nd International Conference on Distributed Computing Systems, pp. 2012. pp. 516 – 525.

[69] Thomas Renner, Lauritz Thamsen and Odej Kao, “Network-aware resource management for scalable data analytics frameworks,” 2015 IEEE International Conference on Big Data (Big Data), 2015, pp. 2793 – 2800.

[70] Xie, J., Yin, S., Ruan, X., Ding, Z., Tian, Y., Majors, J., Manzanares, A., and Qin, X. “Improving MapReduce performance through data placement in heterogeneous Hadoop clusters,”. In Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on (2010), pp. 1–9.

[71] Asmath F. Thaha; Manvir Singh; Anang H. M. Amin; Nazrul M. Ahmad; Subarmaniam Kannan, “Hadoop in OpenStack: Data-location-aware cluster

provisioning," 2014 4th World Congress on Information and Communication Technologies (WICT 2014), 2014. pp. 296 – 301.

[72] Shadi Ibrahim, Hai Jin, Lu Lu, Bingsheng He, Gabriel Antoniu and Song Wu "Maestro: Replica-Aware Map Scheduling for MapReduce," 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2012. pp. 435-442.

[73] Lei Lei, Tianyu Wo and Chunming Hu, "CREST: Towards Fast Speculation of Straggler Tasks in MapReduce," 2011 IEEE 8th International Conference on e-Business Engineering, 2011. pp. 311 – 316.

[74] Cristina L. Abad; Yi Lu; Roy H. Campbell, "DARE: Adaptive Data Replication for Efficient Cluster Scheduling," 2011 IEEE International Conference on Cluster Computing, 2011, pp. 159 – 168.

[75] M. Hammoud, M. S. Rehman, and M. F. Sakr, "Center-of-gravity reduce task scheduling to lower MapReduce network traffic", 2012 IEEE Fifth International Conference on Cloud Computing

[76] Ran Zheng, Genmao Yu, Hai Jin, Xuanhua Shi and Qin Zhang, "Conch: A Cyclic MapReduce Model for Iterative Applications," 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 17-19 Feb. 2016, pp. 264 – 271.

[77] Haiying Shen; Ankur Sarker; Lei Yu; Feng Deng, "Probabilistic Network-Aware Task Placement for MapReduce Scheduling," 2016 IEEE International Conference on Cluster Computing (CLUSTER), 2016, pp, 241 – 250.

[78] Yanfei Guo; Jia Rao; Dazhao Cheng; Xiaobo Zhou. "iShuffle: Improving Hadoop Performance with Shuffle-on-Write," IEEE Transactions on Parallel and Distributed Systems, 2016, Volume: PP, Issue: 99 pp. 1 – 14.

[79] Jingui Li , Xuelian Lin and Xiaolong Cui, "Improving the Shuffle of Hadoop MapReduce," 2013 IEEE 5th International Conference on Cloud Computing Technology and Science, 2-5 Dec. 2013. pp. 266 – 273.

[80] Xiao Yu and Bo Hong "Grouping Blocks for MapReduce Co-Locality," Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International, 25-29 May 2015, pp. 271 – 280.

[81] Muhammad Hanif and Choonhwa Lee, "An efficient key partitioning scheme for heterogeneous MapReduce clusters," 2016 18th International Conference on Advanced Communication Technology (ICACT), 2016, pp. 364-367.

[82] Jianjiang Li, Jie Wu, Xiaolei Yang and Shiqi Zhong "Optimizing MapReduce Based on Locality of K-V Pairs and Overlap between Shuffle and Local Reduce," Parallel Processing (ICPP), 2015 44th International Conference on, 1-4 Sept. 2015. pp. 939 – 948.

[83] Shadi Ibrahim, Hai Jin, Lu Lu, Song Wu and Bingsheng He "LEEN: Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud," 2nd IEEE International Conference on Cloud Computing Technology and Science, 30 Nov.-3 Dec. 2010. pp. 17 – 24.

[84] S. Ancy and M. Maheswari "Locality based data partitioning in Map reduce," 2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT), 2016. pp. 4869 – 4874.

[85] Yaling Xun; Jifu Zhang; Xiao Qin; Xujun Zhao, "FiDoop-DP: Data Partitioning in Frequent Itemset Mining on Hadoop Clusters," IEEE Transactions on Parallel and Distributed Systems, 2017, Volume: 28, Issue: 1, pp. 101 – 114.

[86] Yanrong Zhao, Weiping Wang, Dan Meng, Xiufeng Yang, Shubin Zhang; Jun Li and Gang Guan," A data locality optimization algorithm for large-scale data processing in Hadoop," 2012 IEEE Symposium on Computers and Communications (ISCC), 2012. pp.655 – 661.

[87] S. Seo, I. Jang, K. Woo, I. Kim, J.-S. Kim, and S. Maeng, "Hpmr: Prefetching and pre-shuffling in shared MapReduce computation environment,"in Proceedings of the 2009 IEEE International Conference on Cluster Computing (CLUSTER'07), 2009. pp. 1 – 8.

[88] Aprigio Bezerra, Porfidio Hernández, Antonio Espinosa and Juan Carlos Moure. "Job scheduling in Hadoop with Shared Input Policy and RAMDISK," 2014 IEEE International Conference on Cluster Computing (CLUSTER) 2014, Pages: 355 – 363.

[89] AMD, Hadoop Performance Tuning Guide. Rev 1.0. Retrieved from http://www.admin-magazine.com/HPC/Vendors/AMD/Whitepaper-Hadoop-Performance-Tuning-Guide. 2012.

[90] Intel. Optimizing Hadoop Deployments. Retrieved from https://software.intel.com/sites/default/files/Optimizing%20Hadoop%20Deployments.pdf.

[91] Peng Qin; Bin Dai; Benxiong Huang; Guan Xu,"Bandwidth-Aware Scheduling With SDN in Hadoop: A New Trend for Big Data," IEEE Systems Journal Year: 2015, Volume: PP, Issue: 99 Pages: 1 – 8.

[92] Engin Arslan, Mrigank Shekhar and Tevfik Kosar, " Locality and Network-Aware Reduce Task Scheduling for Data-Intensive Applications," 2014 5th International Workshop on Data-Intensive Computing in the Clouds, 2014. pp. 17 – 24.

[93] Jian Tan, Xiaoqiao Meng and Li Zhang, "Coupling task progress for MapReduce resource-aware scheduling," 2013 Proceedings IEEE INFOCOM. 2013. pp. 1618 – 1626

[94] Mohammad Hammoud and Majd F. Sakr, "Locality-Aware Reduce Task Scheduling for MapReduce," 2011 IEEE Third International Conference on Cloud Computing Technology and Science, 2011. pp. 570 – 576.

[95] Jiahui Jin, Junzhou Luo, Aibo Song, Fang Dong and Runqun Xiong, "BAR: An Efficient Data Locality Driven Task Scheduling Algorithm for Cloud Computing," 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2011. pp. 295 – 304.

[96] Ming-Chang Lee; Jia-Chun Lin; Ramin Yahyapour, "Hybrid Job-Driven Scheduling for Virtual MapReduce Clusters," IEEE Transactions on Parallel and Distributed Systems, 2016, Volume: 27, Issue: 6, pp. 1687 – 1699

[97] Cairong Yan; Xin Yang; Ze Yu; Min Li; Xiaolin Li, "IncMR: Incremental Data Processing Based on MapReduce," 2012 IEEE Fifth International Conference on Cloud Computing, 2012. pp. 534 – 541.

[98] Cloudlab. Retrieved from https://www.cloudlab.us/

[99] Liu, Q., Zhang, N., Yang, X. and Zhu H. Distributed Index Mechanism based on Hadoop, International Symposium on Communications and Information Technologies, pp. 1-7, IEEE, 2014.
[100] Ghoting, A., Kambadur, P., Pednault, E. and Kannan, R. NIMBLE: a toolkit for the implementation of parallel data mining and machine learning algorithms on mapreduce, Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining, KDD' 11, pp. 334-342. ACM, 2011.
[101] Intel. Optimizing Hadoop Deployments. Retrieved from https://software.intel.com/sites/default/files/Optimizing%20Hadoop%20Deployments.pdf
[102] Siliconangle. Wikibon forecasts Big Data market to hit $92.2B by 2026. Retrieved from http://siliconangle.com/blog/2016/03/30/wikibon-forecasts-big-data-market-to-hit-92-2bn-by-2026/
[103] MapR. Retrieved from https://mapr.com/
[104] Yangwook Kang; Yang-suk Kee; Ethan L. Miller; Chanik Park, "Enabling cost-effective data processing with smart SSD", Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on, 6-10 May 2013. pp. 1-12
[105] Md Wasi-ur-Rahman; Nusrat Sharmin Islam; Xiaoyi Lu; Jithin Jose; Hari Subramoni; Hao Wang; Dhabaleswar K. Dk Panda, "High-Performance RDMA-based Design of Hadoop MapReduce over InfiniBand", Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International, 20-24 May 2013, pp. 1908-1917
[106] IBM, "InfoSphere Data Stage". Online available: https://www.ibm.com/ms-en/marketplace/datastage
[107] Oracle, "Data Integrator". Online available: http://www.oracle.com/technetwork/middleware/data-integrator/overview/index.html
[108] Informatica, "PowerCenter". Online available: https://www.informatica.com/products/data-integration/powercenter.html#fbid=uzsZzvdWlvX
[109] Zhenhua Guo, Geoffrey Fox and Mo Zhou. "Investigation of Data Locality in MapReduce", Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on, pp. 419-426, IEEE, 2012.
[110] Peng Zhang, Chunlin Li and Yahui Zhao. "An Improved Task Scheduling Algorithm Based on Cache Locality and Data Locality in Hadoop", Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2016 17th International Conference on, pp. 244-249, IEEE, 2016.
[111] Ruiqi Sun, Jie Yang, Zhan Gao and Zhiqiang He. "A virtual machine based task scheduling approach to improving data locality for virtualized Hadoop", Computer and Information Science (ICIS), 2014 IEEE/ACIS 13th International Conference on, IEEE, 2014.
[112] DigitalOcean. "Introduction to Ganglia on Ubuntu 14.04". Online available: https://www.digitalocean.com/community/tutorials/introduction-to-ganglia-on-ubuntu-14-04

[113] WATERFORD TECHNOLOGIES. “Big Data Statistics & Facts for 2017”. Oniline available: https://www.waterfordtechnologies.com/big-data-interesting-facts/

CURRICULUM VITAE

Sungchul Lee

Email: lsungchul@gmail.com
Homepage: https://lsungchul.wordpress.com

EDUCATION

University of Nevada, Las Vegas **Aug. 2013 – Present**
Ph.D. Student in Computer Science GPA 3.77/4.0
University of Nevada, Las Vegas **May 2012**
M.S. in Computer Science GPA 3.8/4.0
Konkuk University, South Korea **Feb. 2009**
B.S. in Computer Engineering GPA 3.8/4.5

RESEARCH EXPERIENCE

University of Nevada, Las Vegas **Sep. 2013 – Present**
Research Assistant, Center for Energy Research *Las Vegas, NV*
(Funded by the NSF IIA-1301726)

Title: The Solar Energy-Water-Environment Nexus in Nevada

- Development of renewable energy resources is a national priority, and Nevada is aligning its research and development activities in support of this important national goal. The abundant solar flux in Nevada makes it one of the best sources for solar energy generation in the world, and development of this energy source has potential to significantly diversify the economy of the state
- Project size: $20,000,000, composed of over 50 members at University of Nevada, Reno, Desert Research Institute and University of Nevada, Las Vegas
- Team Name: Cyberinfrastructure

Contributions

- Support interdisciplinary renewable energy research, policy, decision-making, outreach and education by using cyberinfrastructure
- Develop integrated data repositories, intelligent and user-friendly software solutions
- Big Data analysis with Nevada Climate data and renewable energy resource
- Large scale of visualization on Nevada Climate Change Portal (NCCP)
- Designed the RESTful web service architecture for NCCP with Java
- Designed Security Algorithm for web service in Nexus Project

University of Nevada, Las Vegas **Jan. 2010 – Dec. 2011**
Research Assistant, Department of Computer Science *Las Vegas, NV*

Title: A Consumer Level Simulation Model for Demand Response Analysis on Smart Grid

- This project advances the current system of Demand-Response by creating a Smart Grid Simulator that allows an intuitive demand response analysis. Substantial amount of electric power can be reduced efficiently by selective demand control over the Smart Grid Simulator.

Contributions

- Designed the user interface and the control software for Simulator with visual C++
- Developed the data structures for simulation models in UML
- Making data anonymization system using RSA encryption algorithm

PUBLICATIONS

Journals

1. Sungchul Lee, Ju-Yeon Jo, and Yoohwan Kim. "Authentication System for Stateless RESTful Web Service." Journal of Computational Methods in Science and Engineering (JCMSE), 2016. Accepted
2. Sungchul Lee, Eunmin Hwang, Ju-Yeon Jo, and Yoohwan Kim. "Big Data Analysis on Personalized Incentive Model with Synthetic Hotel Customer Data." International Journal of Software Innovation, 2016. Vol. 4 Issue 3. pp. 1 - 21
3. Sungchul Lee, Ju-Yeon Jo, and Yoohwan Kim. "Restful Web Service and Web-based Data Visualization for Environmental Monitoring." International Journal of Software Innovation, 2015. Vol. 3. Issue 1. pp. 75-94
4. Sungchul Lee, Ju-Yeon Jo, and Yoohwan Kim. "Environmental Sensor Monitoring with secure RESTful Web Service." International Journal of Services Computing, 2015. Vol. 2 Issue 3. pp. 30-42

Conference

5. Sungchul Lee, Ju-Yeon Jo, and Yoohwan Kim. "Performance Improvement of MapReduce Process by Promoting Deep Data Locality." The 3rd IEEE International Conference on Data Science and Advanced Analytics (DSAA), October 17, 2016. (acceptance rate: 20%)
6. Sungchul Lee, Ju-Yeon Jo, and Yoohwan Kim. "Data Analysis at Single-Mode and Multi-Mode for Data Center." The Twenty Fifth International Conference on Software Engineering and Data Engineering (SEDE), 2016. September 26, 2016
7. Yoohwan Kim, Ju-Yeon Jo, and Sungchul Lee. "A Secure Location Verification Method for ADS-B.", IEEE 35th Digital Avionics Systems Conference. September 25, 2016
8. Sungchul Lee, Ju-Yeon Jo, and Yoohwan Kim. "Secure and Stateless RESTful Web Service Using ID-Based Encryption." 28th International Conference on Computer Applications in Industry and Engineering (CAINE). October 12, 2015
9. Sungchul Lee, Juyeon Jo, and Yoohwan Kim. "A Method for Secure RESTful Web Service." IEEE/ACIS International Conference on Computer and Information Science. June 28, 2015
10. Sungchul Lee, Ju-Yeon Jo, and Yoohwan Kim. "Performance Testing of Web-Based Data Visualization." IEEE International Conference on Systems, Man and Cybernetics (SMC). October 5, 2014
11. Juyeon Jo, Yoohwan Kim, and Sungchul Lee. "Mindmetrics: Identifying users without their login IDs." IEEE International Conference on Systems, Man and Cybernetics (SMC). October 5, 2014
12. Sungchul Lee, Ju-Yeon Jo, Yoohwan Kim, & Haroon Stephen. "A Framework for Environmental Monitoring with Arduino-based Sensors using Restful Web Service." IEEE International Conference on Services Computing, June 27, 2014 (acceptance rate: 20%)

Poster

1. Sungchul Lee, Ju-Yeon Jo, and Yoohwan Kim. "Enhanced Big Data Processing with a

New Hadoop Block Placement Policy.” Research@UNLV Presentation & Tech Expo. Oct 7, 2016

2. Sungchul Lee, Ju-Yeon Jo, and Yoohwan Kim. “Novel Block Placement Policy for Hadoop”, 2016 US-Korea Conference, August 10, 2016
3. Sungchul Lee, Ju-Yeon Jo, and Yoohwan Kim. “Enhanced Big Data Processing with a New Hadoop Block Placement Policy.” 6th Graduate Celebration in UNLV April 25, 2016
4. Sungchul Lee, Ju-Yeon Jo, and Yoohwan Kim. “Advanced MapReduce Process Using Limited Node Block Placement Policy.”, 18th Annual Graduate & Professional Student Research Forum in UNLV. March 12, 2016
5. Sungchul Lee, Ju-Yeon Jo, and Yoohwan Kim. “Advanced MapReduce Process Using Limited Node Block Placement Policy” 2016 Solar Energy-Water-Environment Nexus in Nevada Annual Meeting at UNR March 13, 2016
6. Sungchul Lee, Ju-Yeon Jo, and Yoohwan Kim. “Performance Testing of Web-Based Visualization with Large-Scale Data” 2015 KOCSEA Technical Symposium on December 10-11, Harvey Mudd College, California. December 11, 2015
7. Sungchul Lee, Ju-Yeon Jo, and Yoohwan Kim. “A Framework for Environmental Monitoring with Arduino-based Sensors using Restful Web Service” CWe days 2015 in UNLV. April 8, 2015
8. Sungchul Lee, Ju-Yeon Jo, and Yoohwan Kim. “Performance Testing of Web-Based Data Visualization” CWe days 2015 in UNLV. April 8, 2015
9. Sungchul Lee, Ju-Yeon Jo, and Yoohwan Kim. “Performance Testing of Web-Based Data Visualization” 2015 Graduate & Professional Student Research Forum in UNLV. March 21, 2015
10. Sungchul Lee, Ju-Yeon Jo, and Yoohwan Kim. “A Framework for Environmental Monitoring with Arduino-based Sensors using Restful Web Service” 2014 Graduate & Professional Student Research Forum in UNLV. March 29, 2014
11. Sungchul Lee and Yoohwan Kim. “A consumer level simulation model for demand response analysis on smart grid” College of Engineering: Graduate Celebration. April 27, 2012

Presentation

1. Sungchul Lee, “Deep Data Locality on Hadoop” 2016 KOCSEA Technical Symposium on November 4, 2016
2. Sungchul Lee, “Performance Improvement of MapReduce Process Using Limited Node Block Placement Policy.” The 3rd IEEE International Conference on Data Science and Advanced Analytics (DSAA), October, 17, 2016
3. Sungchul Lee, “Data Analysis at Single-Mode and Multi-Mode for Data Center.” The Twenty Fifth International Conference on Software Engineering and Data Engineering (SEDE), September 27, 2016
4. Sungchul Lee, “ID-Based Authentication for secure and stateless RESTful Web Service.” UNLV Rebel Grad Slam: 3 Minute Thesis Competition, November 4, 2015
5. Sungchul Lee, “Secure and Stateless RESTful Web Service Using ID-Based Encryption.” 28th International Conference on Computer Applications in Industry and Engineering (CAINE). October 12, 2015
6. Sungchul Lee, “A Method for Secure RESTful Web Service.” IEEE/ACIS International Conference on Computer and Information Science. June 29, 2015
7. Sungchul Lee, “IoT on Secure REST Web Service with IDA” UNLV Graduate

Presentation Competition. November 4, 2014

8. Sungchul Lee, “Performance Testing of Web-Based Data Visualization.” IEEE International Conference on Systems, Man and Cybernetics (SMC). October 5, 2014
9. Sungchul Lee, “Mindmetrics: Identifying users without their login IDs.” IEEE International Conference on Systems, Man and Cybernetics (SMC). October 5, 2014
10. Sungchul Lee, “A Framework for Environmental Monitoring with Arduino-based Sensors using Restful Web Service.” 11th IEEE International Conference on Services Computing, June 27, 2014

MASTER THESIS PROJECT EXPERIENCE

University of Nevada, Las Vegas **Aug. 2009**

- Title: Developing a Simulation Model for power demand control analysis and privacy protection in Smart Grid
- Developed the methods, framework and simulator for power demand control and privacy protection in a Smart Grid environment
- Researching the risks of electric usage data.
- Implementing a demo system to be used for handling peak time using Smart Grid

TECHNICAL SKILLS

Language: C, C++, C#, Scala Scheme and Java, Objective-C, XML, PHP, Java Script, NodeJs
Database: Oracle, MySQL, MSSQL, BigData (Hadoop)
Software: Microsoft Visual Studio (6.0, 2005, 2008), Dev C++, GCC, Xcode, UML, R
Hardware: Arduino, Raspberry Pi, Intel Edison
Platform: Linux, MS-DOS/Windows95/98/NT/XP/Vista, Hadoop
Concept: TCP/IP, HTTP, Socket, MFC, Cocoa programming, AWS, RESTful Web Service, Sensor Networking, MapReduce

WORK EXPERIENCE

Rich Robotics **July. 2016 – Aug. 2016**
Software Developer *Las Vegas, NV*

- Developed Database and Server-side code using Node.js
- APWe or SDK development to connect between MS-SQL and RiCH Modules
- Speech Recognizer on Android device for customer engagement

University of Nevada, Las Vegas **Sep. 2013 – Present**
Graduate Assistance *Las Vegas, NV*

- Developed navigation and collision avoidance, algorithms and 3D simulator for Aircraft Systems through simulations studies
- Developed decision making and prediction model using Big data analytic with Hotel Data and NCCP data on Hadoop system
- Developed web-based Mindmetrics system which is identifying users without login IDs
- Developed NextGen Simulator which is air traffic control simulation program

Teaching **July. 2014 – Aug. 2014**
Private Teaching *Las Vegas, NV*

- Java programming for college students teaching grammars and parking system project

Timeless Wisdom, Inc. - Options Trading Export System **May 2012 – Oct. 2013**

Co-founder *Las Vegas, NV*

- Developed Option Trade Strategy and Auto Trading System
- Implemented option pricing models with all option Greeks (esp. Black-Scholes model)
- Analyzed option trading strategies such as Vertical Spread, Time Spread, Iron Condor in relation to VIX
- Designed Software tool to analyze different trading strategies and measure past performances

Designed the trading algorithms and graphical user interface with Java

University of Nevada, Las Vegas **Jan. 2010 – Dec. 2011**
Graduate Assistance *Las Vegas, NV*

- Assistance, and Grading on Computer Organization class, Network class, and Internet Security class

TeleSecurity Sciences, Inc. **Jun. 2010 – Jul. 2010**
Assistant Software Engineer *Las Vegas, NV*

- Developed the encryption and key management system in C for the software product licensing module

Ubiquitous System Technology Corporation **Jul. 2008 – Sep. 2008**
Junior Software Engineer *Seoul, Korea*

- Implement communication server and data converter module with C++ for Indoor system and u-Conference system
- Formulated various services of conference environment on PDA with RFID
- Developed a range of match functions utilizing the information of passengers' location
- Incorporated information on map service of inner-space using the framework of GIS
- Constructed the primary functions of fire alarm and conference monitoring task

Korea Geospatial Information & Communication Co. **Jun. 2007 – Aug. 2007**
Summer Intern *Seoul, Korea*

- Developed GIS (Geographic Information System) database and network
- Provided the raw information on location of internal and external spatial frame using GIS
- Researched and gave presentations on Ruby language

Republic of Korea Air Force (*Mandatory Military Service*) **Dec. 2003 – Mar. 2005**
Signal Corpsman *Osan Air Force Base, Korea*

- Maintained wireless communication equipment (GRC-512/VHF), connecting the radio to Air Base, providing the data about enemy fighters and blue force to air defense force
- Controlled encryption machine and decryption project with captain over RF channel
- Held second-level secret authorization

Tutoring **Dec. 2002 – Oct. 2003 & Oct. 2005 – Dec. 2005**
Private Tutor *Seoul, Korea*

- Math and Science tutor for high school students teaching Pre-cal, Trigonometry, Algebra, Probability/Statistics, Physics, Chemistry, Biology, and Earth Science

ACADEMIC PROJECTS

Genetic Algorithms and Neural Networks (Graduate) **Spring, 2016**

- Implement Back Propagation and Perceptron Neural Networks using C#

Statistical Pattern Recognition (Graduate) **Spring, 2015**

- Calculating image similarity on Hadoop System using MapReduce and LIRE

Compiler Construction (Graduate) **Fall, 2014**

- Constructed a Compiler using Jasmin and Java CUP

Topics in Advanced Computer Science (Adv. Computer Networks - Graduate) **Fall, 2013**

- Implement SCTP Handover simulation using OPNET

Database Management (Adv. Database Management - Graduate) **Spring, 2010**

- Constructed an employee management database using XML

Relational Database (Multi-Paradigm Programming - Graduate) **Fall, 2009**

- Constructed a mini-database using Scheme

RFID Network of Unmanned Vehicles (Bachelor's degree project) **Fall. 2008**

- Implemented an RFID-based road-search module in C under embedded Linux
- Sensing RFID cards to find current position, feeding the movement direction data with RFID, searching the shortest path using Dijkstra algorithm
- Group project with 3 students

Internet Game (Network IWe - Undergraduate) **Fall. 2008**

- Implemented client and server module over TCP/IP in C
- Stored data of character at database

TRAINING

Big Data on Amazon Web Services **Aug. 2014 – Aug. 2014**
Global Knowledge *NV, USA*

- Analyzing Big Data on AWS

Oracle OCP Training, Oracle Korea **Jul. 2008 – Sep. 2008**
Konkuk University and Oracle Korea *Seoul, Korea*

- Constructed database servers using Oracle 10g

RESEARCH GRANT & FUNDED

1. Nexus Student Research Publication & Travel Support **Oct. 2016**
University of Nevada, Las Vegas *Las Vegas, NV*

- Grant: $1,500

2. UNLV Differential Fee **Jan. 2015**
University of Nevada, Las Vegas *Las Vegas, NV*

- Grant: $940

3. GPSA Conference Travel Fund **Apr. 2015**
University of Nevada, Las Vegas *Las Vegas, NV*

- Grant: $575

4. GPSA Conference Travel Fund **May 2014**
University of Nevada, Las Vegas *Las Vegas, NV*

- Grant: $850

5. GPSA Conference Travel Fund **Sep. 2014**
University of Nevada, Las Vegas *Las Vegas, NV*

- Grant: $550

6. The Solar Energy-Water-Environment Nexus in Nevada **Aug. 2013**
National Science Foundation
- Grant Number: IIA-1301726

AWARDS & HONORS

1. Victor & Marjaorie Kunkel Engineering Scholarship **2016-2017**
University of Nevada, Las Vegas *Las Vegas, NV*
- Grant: $4,000

2. Roy & Helen Kelsall Engineering Scholarship **2016-2017**
University of Nevada, Las Vegas *Las Vegas, NV*
- Grant: $1,000

3. UNLV Graduate Access Childcare Scholarship **2016-2017**
University of Nevada, Las Vegas *Las Vegas, NV*
- Grant: $3,000

4. First prize of the scholarship competition of KOCSEA 2015 **Dec. 2015**
KOCSEA Technical Symposium *Claremont, CA.*
- Grant: $500 & Travel supported

5. UNLV Graduate Access Childcare Scholarship **2015-2016**
University of Nevada, Las Vegas *Las Vegas, NV*
- Grant: $5,000

6. GPSA Research Forum Outstanding Presentation **Mar. 2014**
University of Nevada, Las Vegas *Las Vegas, NV*
- Honorable Mention

7. UNLV Access Grant **Jan. 2014**
University of Nevada, Las Vegas *Las Vegas, NV*
- Grant: $2,500

8. UNLV Access Grant **Aug. 2013**
University of Nevada, Las Vegas *Las Vegas, NV*
- Grant: $2,500

9. UNLV Access Grant – Grad NN **Aug. 2011**
University of Nevada, Las Vegas *Las Vegas, NV*
- Grant: $1,000

10. UNLV Access Grant – Grad NN **Jan. 2011**
University of Nevada, Las Vegas *Las Vegas, NV*
- Grant: $2,000

11. Konkuk Shin-Jo Scholarship **Aug. 2008**
Konkuk University *Seoul, Korea*
- Grant: 70/100 of tuition fee (over $4,000)
- Sole winner in the same grade in CS of 2009 (40 students)

REVIEWER

International Conference on Big Data, Cloud Computing and Data Science (BCD) 2016

MEMBERSHIP

IEEE Membership (Student) **2014 – Present**

KSEA membership **2015 – Present**

EXTRA-CURRICULAR ACTIVITIES

English Conversation Club of Konkuk University, Seoul, Korea **2002 – 2009**

- President (Spring, 2007), Vice-President (Fall, 2003)

Magic School, Seoul, Korea **2002 – 2003**

- Vice-President (2002 – 2003)

Calligraphy Club of Konkuk University, Seoul, Korea **2002 – 2003**