Improving Performance of Hadoop Clusters

by

Jiong Xie

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
December 12, 2011

Keywords: MapReduce, Hadoop, Data placement, Prefetching

Copyright 2011 by Jiong Xie

Approved by

Xiao Qin, Chair, Associate Professor of Computer Science and Software Engineering
Cheryl Seals, Associate Professor of Computer Science and Software Engineering
Dean Hendrix, Associate Professor of Computer Science and Software Engineering

Abstract

The MapReduce model has become an important parallel processing model for large-scale data-intensive applications like data mining and web indexing. Hadoop, an open-source implementation of MapReduce, is widely applied to support cluster computing jobs requiring low response time. The current Hadoop implementation assumes that computing nodes in a cluster are homogeneous in nature. Data locality has not been taken into account for launching speculative map tasks, because it is assumed that most map tasks can quickly access their local data. Network delays due to data movement during running time have been ignored in the recent Hadoop research. Unfortunately, both the homogeneity and data locality assumptions in Hadoop are optimistic at best and unachievable at worst, potentially introducing performance problems in virtualized data centers. We show in this dissertation that ignoring the data-locality issue in heterogeneous cluster computing environments can noticeably reduce the performance of Hadoop. Without considering the network delays, the performance of Hadoop clusters would be significatly downgraded. In this dissertation, we address the problem of how to place data across nodes in a way that each node has a balanced data processing load. Apart from the data placement issue, we also design a prefetching and predictive scheduling mechanism to help Hadoop in loading data from local or remote disks into main memory. To avoid network congestions, we propose a preshuffling algorithm to preprocess intermediate data between the map and reduce stages, thereby increasing the throughput of Hadoop clusters. Given a data-intensive application running on a Hadoop cluster, our data placement, prefetching, and preshuffling schemes adaptively balance the tasks and amount of data to achieve improved data-processing performance. Experimental results on real data-intensive applications show that our design can noticeably improve the performance of Hadoop clusters. In summary, this dissertation describes three

practical approaches to improving the performance of Hadoop clusters, and explores the idea of integrating prefetching and preshuffling in the native Hadoop system.

Acknowledgments

I would like to acknowledge and thank the many people whom, without their guidance, friendship and support, this work would not have been possible.

First and foremost, I am thankful to my advisor, Dr. Xiao Qin, for his unwavering support, trust, and belief in me and my work. I would also like to thank him for his advice, guidance, infectious enthusiasm and unbounded energy, even when the road ahead seemed long and uncertain; and Prof. Hendrix for his belief in my work, and for taking the time to serve on my Dissertation Committee. I would also like to thank Dr. Seals for his support, guidance, and advice on all our algorithmic, Mathematical, and Machine Learning questions. I am also grateful to Professor Fa Foster Dai who is the Associate Director of Alabama Microelectronics Sciences and Technology Center, for serving as the university reader.

I have been working with a fantastic research group. I would like to thank my colleagues Xiaojun Ruan, Zhiyang Ding, Shu Yin, Yun Tian, Yixian Yang, Jianguo lu, James Majors and Ji Zhang. All of them have helped me a lot with my research and study; Working with them is beneficial and pleasant. I also appreciate our many discussions and their help in running experiments, sharing their log data, and guiding me through their workloads on many occasions.

I would like to thank the university as a whole for supporting me through three degrees in the Department of Computer Science and Software Engineering and for providing an environment in which excellence is everywhere and mediocrity is not tolerated. Many individuals have provided support, and I want to thank just a few by name, including Yang Qing, and Haiquan Chen for providing variously inspiration, knowledge, and support. I want to thank all of the friends who have helped me and taught me in so many ways, and who put up with

the long hours and the stress that creating a doctoral thesis and building a research career entails. Many thanks to you all. I could not have done this without you.

In addition, I would like to thank my friends in Auburn, including Jiawei Zhang, Ying Zhu, Sihe Zhang, Rui Xu, Qiang Gu, Jingyuan Xiong, Suihan Wu, Min Zheng, and many more. I will value our friendship and miss the time we spent together.

My deepest gratitude goes to my parents Zuobao Xie, and Xiaohua Wang for their years of selfless support. They provided me the basic tools I needed, and then set me free to pursue my goals as I saw them. They quietly provided support in the background and allowed me to look forward.

Most of all, I must thank my girlfriend Fan Yang, whose endless love and encouragement have been my source of inspiration. During the past year, Fan has provided me with sufficient support needed to do research and write this dissertation. I would have never succeeded without her.

Table of Contents

Abstract ii

Acknowledgments iv

List of Figures x

List of Tables xii

1 Introduction 1

1.1 Scope 3

1.1.1 Data Distribution Issue 4

1.1.2 Data Locality Issue 6

1.1.3 Data Transfer Issue 7

1.2 Contribution 8

1.3 Organization 9

2 Background 10

2.1 Mapreduce 11

2.1.1 MapReduce Model 12

2.1.2 Execution Process 13

2.1.3 Scheduler 16

2.2 Hadoop Distributed File System 20

2.2.1 Architecture 20

2.2.2 Execution Process 22

2.2.3 Summary 24

3 Improving MapReduce Performance through Data Placement in Heterogeneous Hadoop Clusters 26

3.1 Motivations for New Data Placement Schemes in Hadoop 26

3.1.1 Data Placement Problems in Hadoop 26
3.1.2 Contributions of our Data Placement Schemes 28
3.1.3 Chapter Organization 28
3.2 The Data Placement Algorithm 28
3.2.1 Data Placement in Heterogeneous Clusters 28
3.2.2 Initial Data Placement 30
3.2.3 Data Redistribution 31
3.3 Implementation of the Data Placement Schemes 32
3.3.1 Measuring Heterogeneity 32
3.3.2 Sharing Files among Multiple Applications 33
3.3.3 Data Distribution. 34
3.4 Performance Evaluation 34
3.5 Summary 40
4 Predictive Scheduling and Prefetching for Hadoop clusters 42
4.1 Motivations for a New Prefetching/Scheduling Mechanism in Hadoop 42
4.1.1 Data Locality Problems in Hadoop 42
4.1.2 Contributions of our Prefetching and Scheduling Mechanism 44
4.1.3 Chapter Organization 45
4.2 Design and Implementation Issues 45
4.2.1 Desgin Challenges 45
4.2.2 Objectives 47
4.2.3 Architecture 47
4.2.4 Predictive Scheduler 49
4.2.5 Prefetching 51
4.3 Performance Evaluation 52
4.3.1 Experimental Environment 52
4.3.2 Individual Node Evaluation 54

4.3.3 Large vs. Small Files 57
4.3.4 Hadoop Clusters 58
4.4 Summary 59
5 Preshuffling 60
5.1 Motivations for a New Preshuffling Scheme 60
5.1.1 Shuffle-Intensive Hadoop Applications 60
5.1.2 Alleviate Network Load in the Shuffle Phase 61
5.1.3 Benefits and Challenges of the Preshuffling Scheme 62
5.1.4 Chapter Organization 63
5.2 Design Issues 63
5.2.1 Push Model of the Shuffle Phase 63
5.2.2 A Pipeline in Preshuffling 64
5.2.3 In-memory Buffer 66
5.3 Implementation Issues in Preshuffling 67
5.4 Evaluation performance 69
5.4.1 Experimental Environment 69
5.4.2 In Cluster 70
5.4.3 Large Blocks vs. Small Blocks 71
5.5 Summary 73
6 Related Work 75
6.1 Implementations of MapReduce 75
6.2 Data Placement in Heterogeneous Computing Environments 76
6.3 Prefetching 77
6.4 Shuffling and Pipline 80
7 Conclusions and Future Work 82
7.1 Conclusions 82
7.1.1 Data distribution mechanism 82

7.1.2 Predictive Scheduling and Prefetching 83
7.1.3 Data Preshuffling 85
7.2 Future Work 87
7.2.1 Small Files 87
7.2.2 Security Issues 88
7.3 Summary 89
Bibliography 91

List of Figures

1.1 A MapReduce application accesses HDFS 5

2.1 The overall process of the word count MapReduce application. 12

2.2 The execution process of the MapReduce programming model. 14

2.3 The HDFS architecture [23] 21

2.4 A client reads data from HDFS [73] 22

2.5 A client writes data to HDFS [73] 23

3.1 Response time of Grep on each node 36

3.2 Response time of Wordcount on each node 36

3.3 Impact of data placement on performance of Grep 37

3.4 Impact of data placement on performance of WordCount 37

4.1 The architecture and workflow of MapReduce 48

4.2 Three basic steps to launch a task in Hadoop. 50

4.3 The execution times of Grep in the native Hadoop system and the prefetching-enabled Hadoop system (PSP). 53

4.4 The execution times of WordCount in the native Hadoop system and the prefetching-enabled Hadoop system (PSP). 54

4.5 The performance of Grep and WordCount when a single large file is processed by the prefetching-enabled Hadoop system (PSP). 55

4.6 The performance of Grep and WordCount when multiple small files are processed by the prefetching-enabled Hadoop system (PSP). 55

4.7 The performance improvement of our prefetching-enabled Hadoop system (PSP) over the native Hadoop system. 56

5.1 The progress trend of WordCount processing 1GB data on the 10-node Hadoop cluster. 70

5.2 Impact of block size on the preshuffling-enabled cluster running WordCount. . . . 72

5.3 Impact of block size on the preshuffling-enabled Hadoop cluster running Sort. . . 72

List of Tables

2.1 The MapReduce functions 13

3.1 The Data Redistribution Procedure 31

3.2 Computing ratios, response times, and number of file fragments for three nodes in a Hadoop cluster 34

3.3 Five Nodes in a Hadoop Heterogeneous Cluster 35

3.4 Computing Ratios of the Five Nodes with Respective of the Grep and WordCount Applications 38

3.5 Six Data Placement Decisions 38

4.1 Test Setting 52

4.2 The Test Sets in Experiments 58

5.1 Test Bed 69

Chapter 1

Introduction

An increasing number of popular applications become data-intensive in nature. In the past decade, the World Wide Web has been adopted as an ideal platform for developing data-intensive applications, since the communication paradigm of the Internet is sufficiently open and powerful. Representative data-intensive Web applications include search engines, online auctions, webmail, and online retail sales, to name just a few. Data-intensive applications like data mining and web indexing need to access ever-expanding data sets ranging from a few gigabytes to several terabytes or even petabytes. Google, for example, leverages the MapReduce model to process approximately twenty petabytes of data per day in a parallel fashion [14]. The MapReduce programming framework can simplify the complexity of running parallel data processing functions across multiple computing nodes in a cluster, because scalable MapReduce helps programmers to distribute programs and have them executed in parallel. MapReduce automatically handles the gathering of results across the multiple machines and return a single result or set. More importantly, the MapReduce platform can offer fault tolerance that is entirely transparent to programmers. Right now, MapReduce is a practical and attractive programming model for parallel data processing in high-performance cluster computing environments.

Hadoop – a popular open-source implementation of the Google's MapReduce model – is primarily developed by Yahoo [23]. Hadoop is used by Yahoo's servers, where hundreds of terabytes of data are generated on at least 10,000 processor cores [78]. Facebook makes use of Hadoop to process more than 15 terabytes of new data per day. In addition to Yahoo and Facebook, a wide variety of websites like Amazon and Last.fm are employing

Hadoop to manage massive amount of data on a daily basis [61]. Apart from Web data-intensive applications, scientific data-intensive applications (e.g., seismic simulations and natural language processing) take maximum benefits from the Hadoop system [9][61].

A Hadoop system basically consists of two major parts. The first part is the Hadoop MapReduce engine – MapReduce [14]. The second component is HDFS – Hadoop Distributed File System [13], which is inspired by Google's GFS (i.e., Google File System). Currently, HDFS divides files into blocks that are replicated among several different computing nodes with no attention to whether the blocks are divided evenly. When a job is initiated, the processor of each node works with the data on their local hard disks. In the initial phase of this dissertation research, we investigate how Hadoop works with its parallel file system, like Lustre. Lustre divides a large file into small pieces, which are evenly distributed across multiple nodes. When the large file is accessed, high aggregated I/O bandwidth can be achieved by accessing the multiple nodes in parallel. The performance of cluster can be improved by Hadoop, because multiple nodes work concurrently to provide high throughput.

Although Hadoop is becoming popular as a high-performance computing platform for data-intensive applications, increasing evidence has shown that performance of data-intensive applications can be severely limited by a combination of a persistent lack of high disk- and network-I/O bandwidth and a significant increase in I/O activities. In other words, performance bottlenecks for data-intensive applications running in cluster environments are caused by disk- and network-I/O rather than CPU or memory performance. There are multiple reasons for this I/O performance problem. First, the performance gap between processors and I/O subsystems in clusters is rapidly widening. For example, processor performance has seen an annual increase of approximately 60% for the last two decades, while the overall performance improvement of disks has been hovering around an annual growth rate of 7% during the same period of time. Second, the heterogeneity of various resources in clusters makes the I/O bottleneck problem even more pronounced.

We believe that there exist efficient ways of improving the performance of Hadoop clusters. The objective of this dissertation work is to investigate offline and online mechanisms, such as data locality and prefetching, to boost performance of parallel data-intensive applications running on Hadoop clusters.

This chapter presents the scope of this research in Section 1.1, highlights the main contributions of this dissertation in Section 1.2, and finally outlines the dissertation organization in Section 1.3.

1.1 Scope

Many large-scale applications are data intensive in nature and require manipulation of huge data sets such as multidimensional scientific data, image files, satellite data, database tables, digital libraries, and the like. MapReduce is an attractive programming model that supports parallel data processing in high-performance cluster computing environments. The MapReduce programming model is highly scalable, because the jobs in MapReduce can be partitioned into numerous small tasks, each of which is running on one computing node in a large-scale cluster [14]. The Hadoop runtime system coupled with HDFS manages the details of parallelism and concurrency to provide ease of parallel programming as well as reinforced reliability [76].

The locality and interdependence issues to be addressed in this study are inherent in large-scale data-parallel computing. Scheduling in MapReduce differs from traditional cluster scheduling in the following two ways [48].

First, the MapReduce scheduler largely depends on data locality, i.e., assigning tasks to computing nodes where input data sets are located. Data locality plays an important role in achieving performance of clusters because the network bisection bandwidth in a large cluster is much lower than the aggregate bandwidth of the disks in computing nodes [14]. Traditional cluster schedulers (e.g., Torque [48]) that give each user a fixed set of computing nodes significantly degrade system performance, because files in Hadoop are distributed

across all nodes as in GFS [66]. Existing schedulers (e.g., Condor [71]) for computational grids address the issue of locality constraints only at the geographic sites level rather than at the computing-node level, because Grids run CPU-intensive applications instead of data-intensive jobs handled by Hadoop. When it comes to a granular fair scheduler, evidence shows that data locality may be problematic for concurrent jobs and small jobs.

Second, the dependence of reduce tasks on map tasks may cause performance problems in MapReduce. For example, reduce tasks cannot be completed until all corresponding map tasks in the job are finished. Such an interdependence issue is not observed in traditional schedulers for cluster computing. The dependence among reduce and map tasks can slow down the performance of clusters by imbalanced workload - some nodes are underutilized and others are overly loaded. A long-running job containing many reduce tasks on multiple nodes will not sitting idle on the nodes until the job's map phases are completed. Therefore, the nodes running idle reduce tasks are underutilized due to the fact that reduce tasks reserve the nodes. To address this performance issue, we propose a preshuffling scheme to preprocess intermediate data between a pair of map and reduce tasks in a long-running job, thereby increasing the computing throughput of Hadoop clusters. We discuss the details of our preshuffling technique in Chapter 5.

Executing data-intensive applications on Hadoop clusters imposes several challenging requirements on resource management design. In particular, we summarize below three main observations for improving performance of Hadoop clusters.

1.1.1 Data Distribution Issue

We observed that data locality is a determining factor for MapReduce performance. To balance workload in a cluster, Hadoop distributes data to multiple nodes based on disk space availability. Such a data placement strategy is very practical and efficient for a homogeneous environment, where computing nodes are identical in terms of computing and disk capacity. In homogeneous computing environments, all nodes have identical workloads, indicating

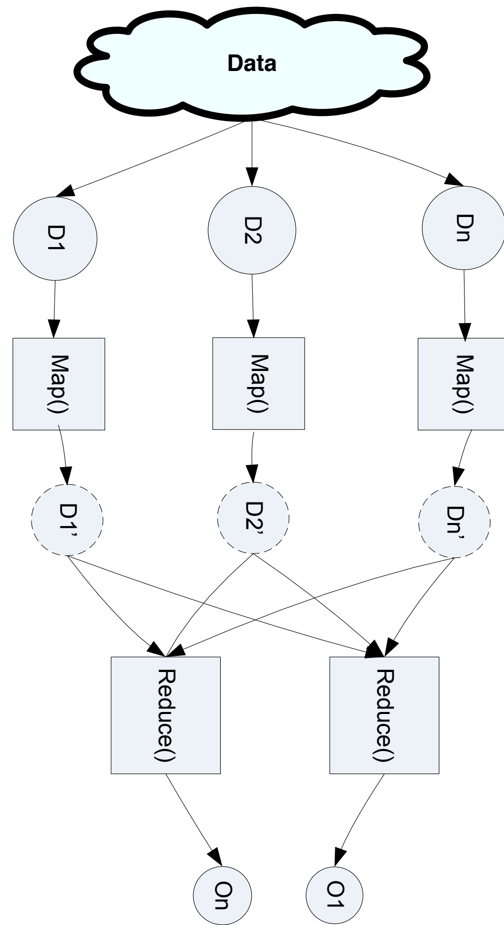


Figure 1.1: A MapReduce application accesses HDFS

that no data needs to be moved from one node to another. In a heterogeneous cluster, however, a high-performance node can complete local data processing faster than a low-performance node. After the fast node finishes processing the data residing in its local disk, the node has to handle the unprocessed data in a slow remote node. The overhead of transferring unprocessed data from slow nodes to fast ones is high if the amount of data moved is large. An approach to improve MapReduce performance in heterogeneous computing environments is to significantly reduce the amount of data moved between slow and fast nodes in a heterogeneous cluster. To balance the data load in a heterogeneous Hadoop cluster, we investigate data placement schemes, which aim to partition a large data set into small fragments being distributed across multiple heterogeneous nodes in a cluster. We explore this data placement issue in Chapter 3).

1.1.2 Data Locality Issue

Our preliminary findings show that CPU and I/O resources in a Hadoop cluster are underutilized when the cluster is running on data-intensive applications. In Hadoop clusters, HDFS is tuned to support large files. For example, typically file sizes in the HDFS file system range from gigabytes to terabytes. HDFS splits large files into several small parts that are distributed to hundreds of nodes in a single cluster; HDFS stores the index information, called meta data, to manage several partitions for each large file. These partitions are the the basic data elements in HDFS, the size of which by default is 64MB. The large block size can shorten disk seek time; however, large block size also causes the transfer time of data access to dominate the entire processing time, making I/O stalls a significant factor in the processing time. The large block size motivates us to investigate prefetching mechanisms that aim to boost I/O performance of Hadoop clusters.

Another factor encouraging us to develop a prefetching mechanism is that ever-faster CPUs are processing data more quickly than the data can be loaded. Simply increasing cache size does not necessarily improve the I/O-subsystem and CPU performance [51]. In

the MapReduce model, before a computing node launches a new application, the application relies on the master node to assign tasks. The master node informs computing nodes what the next tasks are and where the required data blocks are located. The computing nodes do not retrieve the required data and process it until assignment notifications are passed from the master node. In this way, the CPU are underutilized by waiting a long period for the notifications are available from the master node. Prefetching strategies are needed to parallelize these workloads so as to avoid the idle point. The data locality issue is addressed in Chapter 4)

1.1.3 Data Transfer Issue

A Hadoop application running on a cluster can impose heavy network load. This is an especially important consideration when applications are running on large-scale Hadoop clusters. Rapleaf [59] recently encountered this network performance problem, and devised a neat theoretical model for analyzing how network topology affects MapReduce. The two phases of a MapReduce job has two candidates (i.e., shuffling and reduce output) stressing network interconnects in Hadoop clusters. During the shuffle phase, each reduce task contacts every other node in the cluster to collect intermediate files. During the reduce output phase, the final results of the entire job are written to HDFS.

Since results are output to HDFS, the reduce output stage is seemingly the highest intense period of network traffic; however, the shuffle phase has more potential to stress out network interconnects because each node contacts every other node, rather than only two other nodes. Note, however, that the reduce output phase might take longer than the shuffle phase. The average aggregate peak throughput is the aggregate throughput at which some component in the network resource saturates (i.e., when the network is at the maximum throughput capacity). Once one component in the network saturates, the job as a whole won't be able to go any faster even if there are other underutilized computing nodes.

Moreover, Hadoop clusters are supported, in many cases, by virtual machines and transfer massive amount of processing data between map and reduce tasks through the clusters' network interconnects. When the Hadoop clusters scale up, the network interconnects becomes the I/O bottleneck of the clusters. We address this performance issue in Chapter 5.

1.2 Contribution

To remedy aforementioned performance problems in Hadoop clusters, our dissertation research investigates data placement strategies, prefetching and preshuffling schemes that are capable of reducing data movement activities and improving throughput of the Hadoop cluster. In what follows, we list a set of key contributions of the dissertation.

- **Data placement in HDFS.** We develop a data placement mechanism in the Hadoop distributed file system or HDFS to initially distribute a large data set to multiple computing nodes in accordance with the computing capacity of each node. More specifically, we implement a data reorganization algorithm in HDFS in addition to the data redistribution algorithm. The data reorganization and redistribution algorithms implemented in HDFS can be used to solve the data skew problem, which arises due to dynamic data insertions and deletions.

- **Online prefetching mechanism.** We observe the data movement and task process patterns in Hadoop clusters, and design a prefetching mechanism to reduce data movement activities during running time to improve the clusters' performance. We show how to aggressively search for the next block to be prefetched, thus avoiding I/O stalls incurred by data accesses. At the core of our approach is a predictive scheduling module and prefetching algorithm.

- **Preshuffle.** We propose a shuffle strategy for Hadoop to reduce network overhead caused by data transfer. When intermediate data is ready, the reshuffling modules identify and transfer the data to destinations as soon as possible.

1.3 Organization

Before starting with the main topic of this dissertation, we give a general introduction on Mapreduce and Benckmarks in Chapter 2.

In Chapter 3, we study a data placement mechanism in the HDFS file system to distribute initial large data sets to multiple computing nodes in accordance with the computing capacity of each node.

In Chapter 4, we present data movement and task process patterns of Hadoop. In addition, we design a prefetching mechanism to reduce the amount of data transferred through network interconnects during running time.

In Chapter 5, we propose a shuffle strategy incorporated into Hadoop to reduce network overhead imposed by data transfers.

We present prior studies related to this dissertation research in Chapter 6.

Finally, Chapter 7 summarizes the main contributions of this dissertation and indicates on future directions for this research.

Chapter 2

Background

The MapReduce programming model simplifies the complexity of running parallel data processing functions across multiple computing nodes in a cluster, by allowing a programmer with no specific knowledge of parallel programming to create MapReduce functions running in parallel on the cluster. MapReduce automatically handles the gathering of results across the multiple nodes and returns a single result or set. More importantly, the MapReduce runtime system offers fault tolerance that is entirely transparent to programmers [14].

Hadoop, a popular open-source implementation of Google's MapReduce model, is developed primarily by Yahoo [23]. The Apache Hadoop system is a distributed MapReduce project of the Apache Foundation implemented in the java programming language. As Hadoop is published under the Apache License, Hadoop's source code is publicly available for download. Hadoop is deployed in Yahoo's servers, where hundreds of terabytes of data are generated on at least 10,000 cores [78]. Facebook makes use of Hadoop clusters to process more than 15 terabytes of new data per day. In addition to Yahoo and Facebook, other web giants such as Amazon and Last.fm are employing Hadoop clusters to manage massive amounts of data on a daily basis [61]. Apart from data-intensive web applications, scientific data-intensive applications (e.g., seismic simulation and natural language processing) take maximum benefits from the Hadoop system.

The Hadoop system has two core component. The first component is a distributed file system called HDFS (see Section 2.1); the second one is a MapReduce programming framework (i.e., runtime system) for processing large datasets (see Section2.2). Higher levels in the software stack consists of (1) Pig [26] and Hive [25], user-friendly parallel data processing languages, (2)Zoomkeeper [27] a high-availability directory and configuration service, and

(3) HBase [24], a web-scale distributed column-oriented store designed after its proprietary predecessors [7][10].

In this dissertation research, we pay attention to the two core elements of Hadoop - HDFS and Hadoop runtime system. The basic Hadoop is composed of Hadoop runtime system, an implementation of MapReduce designed for large clusters, and the Hadoop Distributed File System (i.e., HDFS), a file system optimized for batch-oriented workloads like data analysis applications. In most Hadoop jobs, HDFS is used to store both the input of map tasks and the output of reduce tasks. Our design and implementation presented in this dissertation can be easily integrated with any distributions of Hadoop, and our techniques integrated in Hadoop can also be easily employed by high levels in the Hadoop software stack.

This chapter features two sections. Section 2.1 give the background of the google's MapReduce programming model along with the usage of MapReduce. Section 2.2 discusses the Hadoop Distributed File System (HDFS) and briefly describes how does HDFS work.

2.1 Mapreduce

MapReduce is a programming model and an associated implementation for processing and generating large data sets [14]. The MapReduce model was designed for unstructured data processed by large clusters of commodity hardware; the functional style of MapReduce automatically parallelizes and executes large jobs over a computing cluster. The MapReduce model is capable of processing many terabytes of data on thousands of computing nodes in a cluster. MapReduce automatically handles the messy details such as handling failures, application deployment, task duplications, and aggregation of results, thereby allowing programmers to focus on the core logic of applications.

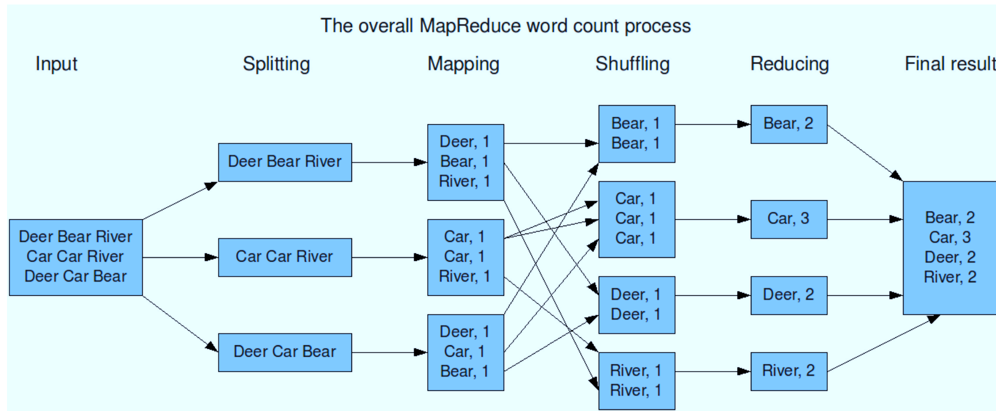


Figure 2.1: The overall process of the word count MapReduce application.

2.1.1 MapReduce Model

Each MapReduce application has two major types of operations - a map operation and a reduce operation. MapReduce allows for parallel processing of the map and reduction operations in each application. Each mapping operation is independent of the others, meaning that all mappers can be performed in parallel on multiple machines. In practice, the number of concurrent map operations is limited by the data source and/or the number of CPUs near that data. Similarly, a set of reduce operations can be performed in parallel during the reduction phase. All outputs of map operations that share the same key are presented to the same reduce operation. Although the above process seemingly inefficient compared to sequential algorithms, MapReduce can be applied to process significantly larger datasets than "commodity" servers. For example, a large computing cluster can use MapReduce to sort a petabyte of data in only a few hours. Parallelism also offers some possibility of recovering from partial failure of computing nodes or storage units during the operation. In other words, if one mapper or reducer fails, the work can be rescheduled, assuming the input data is still available. Input data sets are, in most cases, available even in presence of storage unit failures, because each data set normally has three replicas stored in three individual storage unites.

A MapReduce program has two major phases - a map phase and a reduce phase. The map phase applies user specified logic to input data. The results, called as intermediate results, are then fed into the reducer phase so the intermediate results can be aggregated and written as a final result. The input data, intermediate result, and final result are all represented in the key/value pair format [39].

Figure 2.2 shows an executional example of the MaReduce model. As shown by the diagram during their respective phases multiple map and reduce jobs are executed in parallel in multiple computing nodes. MapReduce is also usually described in form of the following functions summarized in Table 2.1.

Table 2.1: The MapReduce functions

map (k1,v1) ¿¿ list(k2,v2)
reduce (k2,list(v2)) ¿¿ list(k3,v3)

It is worth noting that the map phase must transform input data into intermediate data from which the reduce phase can gather and generate final results.

2.1.2 Execution Process

The MapReduce process can be divided into two parts, namely, a Map section and a Reduce section. Figure 2.2 shows a diagram representing the execution process of the MapReduce model.

Map Task Execution

Each map task is assigned a portion of an input file called a split. By default, a split contains a single HDFS block with 64MB, and the total number of file blocks normally determines the number of map tasks.

The execution of a map task can be separated into two stages. First, the map phase reads the task's split and organizes the split into records (key/value pairs), and applies the map function to each record. After the map function has been applied to each input record,

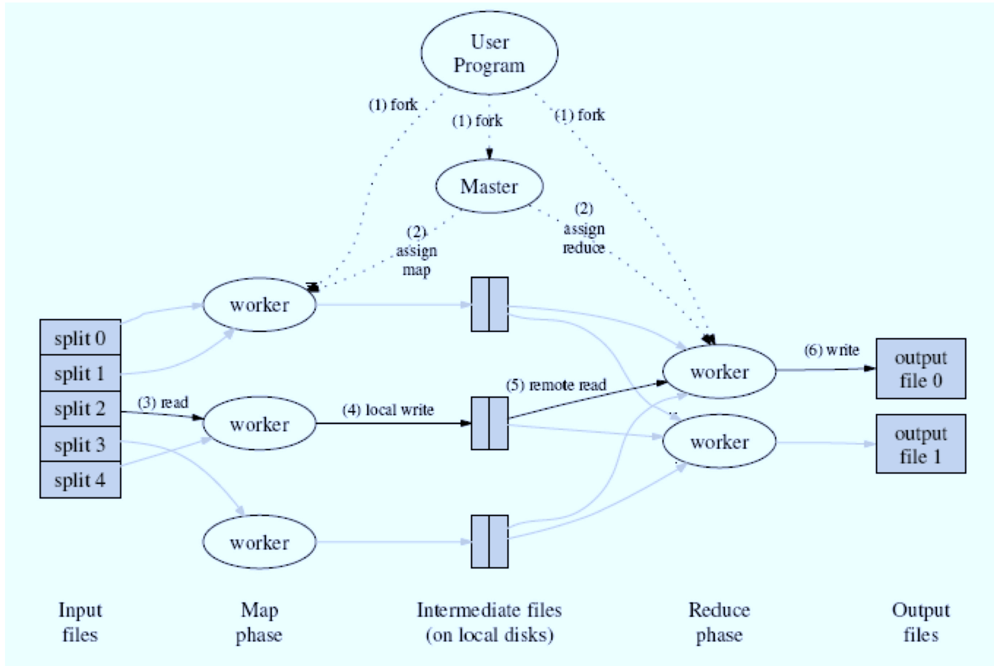


Figure 2.2: The execution process of the MapReduce programming model.

the commit phase registers the final output with the TaskTracker, which then informs the JobTracker that the task has been completed. The output of the map step is consumed by the reduce step, so the OutputCollector stores map output in a format that is easy for the reduce tasks to consume. Intermediate keys are assigned to reducers by applying a partitioning function. Thus, the OutputCollector applies this function to each key produced by the map function, and stores each record and partition number in an in-memory buffer. The OutputCollector spills this information to a disks when a buffer reaches its capacity.

A spill of the in-memory buffer involves sorting the records in the buffer first by partition number, then by key. The buffer content is written to a local file system as a data file and index file. This points to the offset of each partition in the data file. The data file contains the records, which are sorted by the key within each partition segment.

During the commit phase, the final output of a map task is generated by merging all the spill files produced by this map task into a single pair of data and index files. These files are registered with the TaskTracker before the task is completed. The TaskTracker reads these files to service requests from reduce tasks.

Reduce Task Execution

The execution of the reduce task contains three steps.

- In the shuffle step, the intermediate data generated by the map phase is fetched. Each reduce task is assigned a partition of the intermediate data with a fixed the key range, so the reduce task must fetch the content of this partition from every map task's output in the cluster.
- In the sort step, records with the same key are grouped together to be processed by the next step.
- In the reduce step, the user-defined reduce function is applied to each key and corresponding list of values.

In the shuffle step, a reduce task fetches particular data from each map task. The Job-Tracker relays the location of every TaskTracker that hosts a map output to every TaskTracker that is executing a reduce task. Note that a reduce task cannot fetch the output of a map task until the map has finished its execution and commitment of its final output to the disk.

After receiving partitions from all mappers' outputs, the reduce task enters the sort step. The output generated from mappers for each partition is already sorted by the reduce key. The reduce task merges these runs together to produce a single run that is sorted by key. The task then enters the last reduce step, in which the user-defined reduce function is invoked for each distinct key in a sorted order, passing it the associated list of values. The output of the reduce function is written to a temporary location on HDFS. After the reduce function has been applied to each key in the reduce task's partition, the task's HDFS output file is automatically renamed from its temporary location to its final location.

In this design, outputs of both map and reduce tasks are written to disks before the outputs can be consumed. This output writing process is particularly expensive for reduce tasks,

because their outputs are written to HDFS. Output materialization simplifies fault tolerance, because such outputs reduce the number of states that must be restored to consistency after any node failure. If any task (regardless of mappers or reducers) fails, the JobTracker simply schedules a new task to perform the same work assigned to the failed task. Since a task never exports any data other than its final results, no further fault-tolerant actions are needed.

2.1.3 Scheduler

To better the limitations of current Hadoop schedulers that motivate us to develop our solutions, let us explain some key concepts used in Hadoop scheduling.

In a Hadoop cluster, there is a central scheduler managed by a master node, called JobTracker. Worker nodes, called TaskTrackers, are responsible for task executions. JobTracker is responsible not only for tracking and managing machine resources across the cluster, but also for maintaining a queue of currently running MapReduce Jobs. Every TaskTracker periodically reports its state to the JobTracker via a heartbeat mechanism. TaskTrackers concurrently execute the task in each slave nodes.

The JobTracker and TaskTracker transfer information through a heartbeat mechanism. The TaskTracker sends a message to the JobTracker once every specified intervals (e.g., a message every 2 seconds). The Heartbeat mechanism provides a communication channel between the JobTracker and the TaskTracker. A task assignment is delivered to the TaskTracker in the form of a heartbeat. If the task fails, the JobTracker can keep track of this failure because the JobTracker receives no reply from the TaskTracker. The JobTracker monitors the heartbeats received from the TaskTracker to make the task assignment decisions. If a heartbeat is not received from a TaskTracker during a specified time period, the TaskTracker is assumed to be malfunction. In this case, the JobTracker will relaunch all tasks that previously assigned to this failed TaskTracker in another functioning node.

The default Hadoop use the FIFO (i.e., First-In-First-Out) policy based on 5 optional scheduling priorities to schedule jobs from a work queue. In the following subsections, let us introduce a few popular scheduling algorithms widely employed in Hadoop clusters.

FAIR Scheduler

Zaharia et. al. [22] proposed the FAIR scheduler, optimized for multi-user environments, in which a single cluster is shared among a number of users. The FAIR algorithm is used in the data mining research field to analyze log files. The FAIR scheduler aims to reduce idle times of short jobs, thereby offering fast response times of the short jobs.

The scheduler in Hadoop organizes jobs into pools, among which resources are shared. Each pool is assigned a guaranteed minimum share, which ensures that certain users or applications always get sufficient resources. Fair sharing can also work with job priorities, which are used as weights to determine the fraction of total compute time allocated to each job. Fair scheduling assigns resources to jobs so that all jobs consume, on average, an equal share of resources.

The Fair Scheduler allows all jobs to run by a default or specified configuration file, which limits the number of jobs per user and per pool. This configuration file is very useful in two particular cases. First, a user attempts to submit hundreds of jobs at once. Second, many simultaneously running jobs cause high context-switching overhead and an enormous amount of intermediate data. Limiting the number of running jobs, of course, does not cause any subsequently submitted jobs to fail. However, newly arrived jobs must wait in the sheduler's queue until some of the running jobs are completed.

Capacity Scheduler

The capacity scheduler [23] is a scheduler developed by Yahoo for large resource-sharing clusters. Submitted jobs are organized and placed in multiple queues, each of which is guaranteed to access a fraction of a cluster's capacity (i.e., number of task slots). All jobs

submitted to a given queue have access to the resources guaranteed for that queue. If tasks of jobs in queues have excess capacity, the tasks are killed. Free resources can be allocated to any queue beyond its capacity. When there is demand for the resources of queues running below capacity at a future point in time (e.g., tasks scheduled on the resources complete), the resources will be assigned to jobs on queues running below the capacity. If inactive queues start acquiring job submissions, their lost capacity will be reclaimed.

Queues can also support job priorities, which are disabled by default. In a queue, jobs with higher priority have access to the queue's resources prior to the access of jobs with lower priority. However, once a job is running regardless of its priority, the job will not be preempted by any higher priority job. Nevertheless, new tasks from the higher priority job will be preferentially scheduled in the queue. In order to prevent one or more users from monopolizing resources, each queue enforces a limit on the percentage of resources allocated to a user at any given time, if all the users are using the resources.

Whenever a TaskTracker is free, the capacity scheduler chooses a queue with the most free resources. Once the queue is selected, the scheduler picks a job in the queue according to the job's priority. This scheduling mechanism ensures that there is enough free main memory in the TaskTracker to run the job's task in the event that the job has large memory requirements. In this way, the resource requirement of the task can always be promptly honored.

LATE Scheduler

The LATE (i.e., Longest Approximate Time to End) scheduler [47] improves response time of Hadoop clusters in multiuser environments by improving speculative executions. The default speculative execution is a mechanism that rebalances the load on worker nodes and reduces response time by relaunching slow tasks on other TaskTrackers with more computing resources. In this way, slow tasks are duplicated on machines with free slots. This LATE scheduler achieves good utilization when jobs are about to end. LATE also reduces execution

time of slow tasks. LATE counters effects of overload by multiple task assignments on fast machines in heterogeneous cluster computing environments.

Unlike the native speculative execution algorithm, the LATE scheduler focuses on expected time left and relaunches tasks expected to "finish farthest into the future". To better accommodate different types of tasks, a task progress is divided into zones. A user defined limit is used to control the number of speculative tasks assigned to one node. The LATE scheduler shortens response times in heterogeneous Hadoop clusters.

Dynamic Priority Scheduler

The idea of dynamic priority scheduling is to assign priorities based on temporal parameters and maximize resource utilization. A dynamic priority scheduler adapts to dynamically changing progress and forms an optimal configuration in a self-sustained manner. The dynamic scheduler [21] based on the Tycoon [38] system implemented in Hadoop uses a market based approach for task assignments. Each user is given an initial amount of virtual currency. While submitting a job, users can declare a certain spending rate per unit time. The scheduler chooses jobs that earn the maximum "money" for the scheduler. Users can adjust their spending rate in order to change the priorities of their jobs. The users can independently determine their jobs' priorities; the system allocates running time according to spending rate. If the account balance of a user reaches zero, no further tasks of that user will be assigned.

Hadoop on Demand [84] uses the above existing algorithms to manage resources in Hadoop clusters. The FAIR scheduler and capacity scheduler are adopted to achieve fairness; the LATE scheduler is deployed to reduce job response times. The dynamic priority scheduler is employed to achieve adjustable priorities based on dynamically changing progresses [57][34].

2.2 Hadoop Distributed File System

The Hadoop Distributed File System or HDFS is a distributed file system designed to run on commodity hardware. HDFS is the primary distributed storage used by Hadoop applications on clusters. Athough HDFS has many similarities with existing distributed file systems, the differences between HDFS and other systems are significant. For example, HDFS is highly fault-tolerant and is designed to be deployed on cost-effective clusters. HDFS - offering high throughput access to application data - is suitable for applications that have large data sets. HDFS relaxes several POSIX requirements to enable streaming access to file system data. HDFS is not fully POSIX compliant, because the requirements for a POSIX file system differ from the design goals of Hadoop applications. HDFS trades fully POSIX compliance for increased data throughput, since HDFS was designed to handle very large files.

2.2.1 Architecture

HDFS uses a master-slave architecture, in which a master is called NameNode and slaves are referred to as DataNodes. Figure 2.3 shows a diagram representing the architecture of HDFS. Basically, an HDFS cluster consists of a single NameNode, which manages the file system namespace and regulates access of clients to files. In addition, there are a number of DataNodes. Usually, each node in a cluster has one DataNode that manages storage of the node on which tasks are running. HDFS exposes file system namespace and allows user data to be stored in files. Internally, a file is split into one or more blocks stored in a set of DataNodes. The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. The NameNode also determines the mappings of blocks to DataNodes. The DataNodes not only are responsible for serving read and write requests issued from the file system's clients, but also perform block creation, deletion, and replication upon instructions from the NameNode.

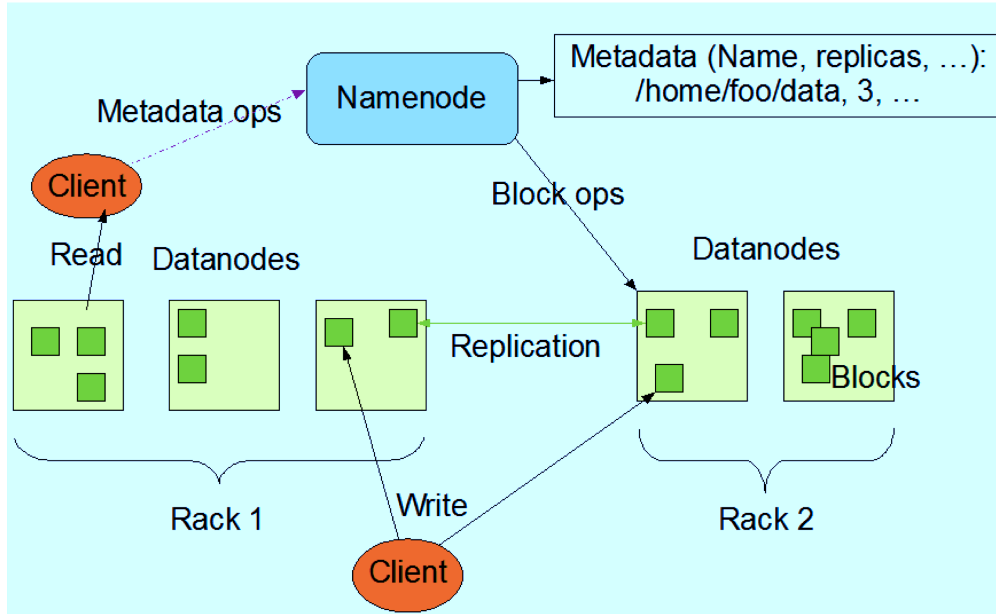


Figure 2.3: The HDFS architecture [23]

The NameNode and DataNode are software modules designed to run on a GNU/Linux operating system. HDFS is built using the Java language as any machine that supports Java can run the NameNode or DataNode modules. Usage of the highly portable Java language means that HDFS can be employed on a wide range of machines. A typical deployment has a dedicated machine that runs only the NameNode module. Each of the other machines in the cluster runs one instance of the DataNode module. The HDFS architecture does not preclude running multiple DataNodes on the same machine, but in reality this is rarely the case.

The existence of a single NameNode in a cluster greatly simplifies the architecture of HDFS. The NameNode is an arbitrator and repository for all metadata in HDFS. The HDFS system is designed in such a way that user data never flows through the NameNode.

HDFS is designed to support very large files, because Hadoop applications are dealing with large data sets. These Hadoop applications write their data only once but read the data one or more times and require these reads to be performed at streaming speeds. HDFS supports write-once-read-many semantics on files. A typical block size used by HDFS is

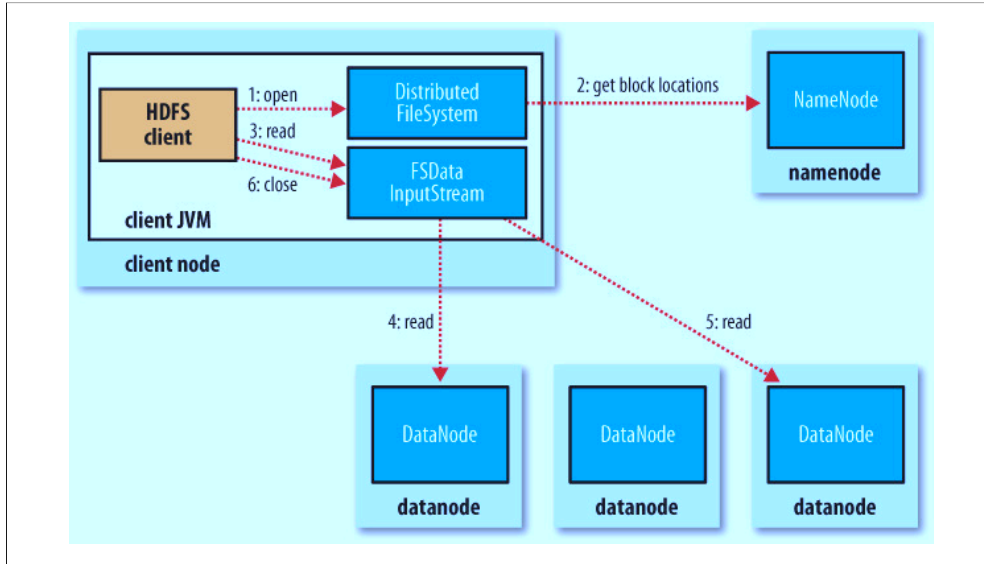


Figure 2.4: A client reads data from HDFS [73]

64 MB; thus, an HDFS file is chopped up into 64 MB chunks. If it is possible, chunks are residing on different DataNodes .

2.2.2 Execution Process

This section introduces the execution process in HDFS. We describe the reading process followed by the writing process.

File read

Figure 2.4 depicts the reading process from HDFS. In the first step, the client opens a file by calling open() on the FileSystem object, which is an instance of DistributedFileSystem in HDFS. Next, the DistributedFileSystem calls the NameNode in HDFS to determine the locations of the first few blocks of the file. For each block NameNode returns the addresses of the Datanaodes storing a copy of that block. The DistributedFileSystem returns an FSDataInputStream to the client from which data is retrieved. FSDataInputStream wraps a DFSInputStream, which manages the DataNode and NameNode I/O. After this step, the client calls read() on the stream DFSInputStream, which has stored the DataNode addresses

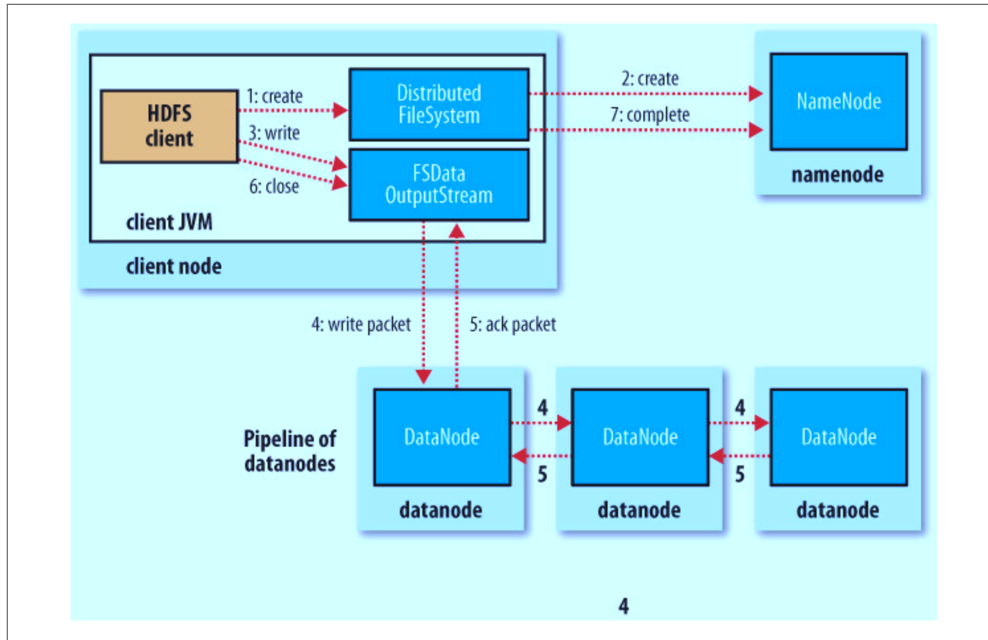


Figure 2.5: A client writes data to HDFS [73]

for the first few blocks in the file. The client connects to the first (closest) DataNode for the first block in the file. Then, data is streamed from the DataNode to the client repeatedly invoking read() on the stream. When the end of the block is reached, DFSInputStream will terminate the connection with the DataNode, and find the best DataNode for the next block. The above steps are transparent to the client. Finally, DFSInputStream calls close() on the FSDataInputStream if the client has finished reading the data.

File write

Figure 2.5 depicts the seven-step writing process in HDFS. First, the client creates a file by calling create() on DistributedFileSystem. Second, DistributedFileSystem makes an RPC call to NameNode to create a brand new file containing no blocks in the filesystem's namespace. NameNode needs to ensure that the new file has not been created before. If the file-existence check is passed, DistributedFileSystem returns a FSDataOutputStream back to the client to start writing data to the file system. FSDataOutputStream wraps a DFSOutput Stream that handles communication between DataNodes and NameNode.

Next, the client begins writing data. DFSOutputStream splits data into packets, which are written to an internal queue called the data queue. The data queue is consumed by the Data Streamer. Fourth, the DataStreamer streams the packets to the first DataNode in a pipeline, which stores the packets and forwards them to the second DataNode in the pipeline. The second DataNode stores the packet and forwards it to the third (and last) DataNode. Please note that each data block has three replicas stored in three different DataNodes. Fifth, the DFSOutputStream maintains an internal queue that called the ACK queue contains packets waiting to be acknowledged by the three DataNodes storing the three replicas of the data block. A packet is removed from the ACK queue only when acknowledgements are received from all the three DataNodes in the pipeline. Sixth, when the client has finished writing data, close() is invoked on the stream to flush all the remaining packets to the DataNode pipeline and waits for corresponding acknowledgements. Finally, the client contacts NameNode to signal that the writing process is complete.

2.2.3 Summary

An advantage of using the Hadoop Distributed File System or HDFS is data awareness between JobTracker and TaskTracker. JobTracker schedules map/reduce jobs to TaskTrackers with an awareness of data locations. For example, let us consider a case where node A contained data (x,y,z) and node B contained data (a,b,c). JobTracker schedules node B to perform map/reduce tasks on (a,b,c) and node A perform map/reduce tasks on (x,y,z). This scheduling reduces the amount of traffic over the network and prevents unnecessary data transfer. When Hadoop is used in combination with other file systems, this data location awareness may not be supported by the other file systems, which can have a significant negative impact on the performance of Hadoop jobs processing massive amount of data.

A limitation of HDFS is that it cannot be directly mounted by an existing operating system. Transferring data into and out from HDFS are often performed before and after executing a Hadoop application. Such a data transferring process can be inconvenient and

time consuming. A File system in Userspace (FUSE) virtual file system (VFS) has been developed to address this problem for Linux and other Unix systems. The purpose of a VFS is to allow client applications to access different types of file systems in a uniform way.

Chapter 3

Improving MapReduce Performance through Data Placement in Heterogeneous Hadoop Clusters

3.1 Motivations for New Data Placement Schemes in Hadoop

3.1.1 Data Placement Problems in Hadoop

An increasing number of popular applications have become data-intensive in nature. In the past, the World Wide Web has been adopted as an ideal platform for developing data-intensive applications, since the communication paradigm of the Web is sufficiently open and powerful. Representative data-intensive Web applications include, but not limited to, search engines, online auctions, webmails, and online retail sales. Data-intensive applications like data mining and web indexing need to access ever-expanding data sets ranging from a few gigabytes to several terabytes or even petabytes. Google, for example, leverages the MapReduce model to process approximately twenty petabytes of data per day in a parallel fashion [14]. MapReduce is an attractive model for parallel data processing in high-performance cluster computing environments. The scalability of MapReduce is proven to be high, because a MapReduce job is partitioned into numerous small tasks running on multiple machines in a large-scale cluster.

As description in Chapter 2.1, a MapReduce application directs file queries to a namenode, which in turn passes the file requests to corresponding data nodes in a cluster. Then, the data nodes concurrently feed Map functions in the MapReduce application with large amounts of data. When new application data are written to a file in HDFS, fragments of a large file are stored on multiple data nodes across a Hadoop cluster. HDFS distributes file

fragments across the cluster, assuming that all the nodes have identical computing capacity. Such a homogeneity assumption can potentially hurt the performance of heterogeneous Hadoop clusters. Native Hadoop makes the following assumptions. First, it is assumed that nodes in a cluster can perform work at roughly the same rate. Second, all tasks are assumed to make progress at a constant rate throughout time. Third, There is no cost to launching a speculative task on a node that would otherwise have an idle slot. Fourth, tasks in the same category (i.e., map or reduce) require roughly the same amount of work. These assumptions motivate us to develop data placement schemes that can noticeably improve the performance of heterogeneous Hadoop clusters.

We observe that data locality is a determining factor for Hadoop's performance. To balance workload, Hadoop distributes data to multiple nodes based on disk space availability. Such data placement strategy is very practical and efficient for a homogeneous environment where nodes are identical in terms of both computing and disk capacity. In homogeneous computing environments, all the nodes have identical workload, assuming that no data needs to be moved from one node into another. In a heterogeneous cluster, however, a high-performance node tends to complete local data processing faster than a low-performance node. After the fast node finishes processing data residing in its local disk, the node has to handle unprocessed data in a remote slow node. The overhead of transferring unprocessed data from slow nodes to fast peers is high if the amount of moved data is huge. An approach to improve MapReduce performance in heterogeneous computing environments is to significantly reduce the amount of data moved between slow and fast nodes in a heterogeneous cluster. To balance data load in a heterogeneous Hadoop cluster, we are motivated to investigate data placement schemes, which aim to partition a large data set into data fragments that are distributed across multiple heterogeneous nodes in a cluster.

3.1.2 Contributions of our Data Placement Schemes

In this chapter, we propose a data placement mechanism in the Hadoop distributed file system or HDFS to initially distribute a large data set to multiple nodes in accordance to the computing capacity of each node. More specifically, we implement a data reorganization algorithm in addition to a data redistribution algorithm in HDFS. The data reorganization and redistribution algorithms implemented in HDFS can be used to solve the data skew problem due to dynamic data insertions and deletions.

3.1.3 Chapter Organization

The rest of the Chapter is organized as follows. Section 3.2 describes the data distribution algorithm. Section 3.3 describes the implementation details of our data placement mechanism in HDFS. In Section 3.4, we present the evaluation results and Section 3.5 summarizes the design and implementation of our data placement scheme for heterogeneous Hadoop clusters.

3.2 The Data Placement Algorithm

3.2.1 Data Placement in Heterogeneous Clusters

In a cluster where each node has a local disk, it is efficient to move data processing operations to nodes to which application data are located. If data are not locally available in a processing node, data have to be moved via network interconnects to the node that performs the data processing operations. Transferring a large amount of data leads to excessive network congestions, which in turn can deteriorate system performance. HDFS enables Hadoop applications to transfer processing operations toward nodes storing application data to be processed by the operations.

In a heterogeneous cluster, the computing capacities of nodes may significantly vary. A high-performance node can finish processing data stored in a local disk of the node much

faster than its low-performance counterparts. After a fast node completes the processing of its local input data, the fast node must perform load sharing by handling unprocessed data located in one or more remote slow nodes. When the amount of transferred data due to load sharing is very large, the overhead of moving unprocessed data from slow nodes to fast nodes becomes a critical performance bottleneck in Hadoop clusters. To boost the performance of Hadoop in heterogeneous clusters, we aim to minimize data movement activities observed among slow and fast nodes. This goal can be achieved by a data placement scheme that distributes and stores data across multiple heterogeneous nodes based on their computing capacities. Data movement overheads can be reduced if the number of file fragments placed on the disk of each node is proportional to the node's data processing speed.

To achieve the best I/O performance, one may make replicas of an input data file of a Hadoop application in a way that each node in a Hadoop cluster has a local copy of the input data. Such a data replication scheme can, of course, minimize data transfer among slow and fast nodes in the cluster during the execution of the Hadoop application. Unfortunately, such a data-replication approach has three obvious limitations. First, it is very expensive to create a large number of replicas in large-scale clusters. Second, distributing a huge number of replicas can wastefully consume scarce network bandwidth in Hadoop clusters. Third, storing replicas requires an unreasonably large amount of disk space, which in turn increases the cost of building Hadoop clusters.

Although all replicas can be produced before the execution of Hadoop applications, significant efforts must be make to reduce the overhead of generating excessive number of replicas. If the data-replication approach is employed in Hadoop, one has to address the problem of high overhead for creating file replicas by implementing a low-overhead file-replication mechanism. For example, Shen and Zhu developed a proactive low-overhead file replication scheme for structured peer-to-peer networks [67]. Shen and Zhu's scheme may be incorporated to overcome this limitation.

To address the above limitations of the data-replication approach, we are focusing on data-placement strategies where files are partitioned and distributed across multiple nodes in a Hadoop cluster without any data replicas. Our data placement approach does not rely on any comprehensive scheme to deal with data replicas. Nevertheless, our data placement scheme can be readily integrated with any data-replication mechanism.

In our data placement management mechanism, we designed two algorithms and incorporated the algorithms into Hadoop's HDFS. The first algorithm is to initially distribute file fragments to heterogeneous nodes in a cluster (see Section 3.2.2). When all file fragments of an input file required by computing nodes are available in a node, these file fragments are distributed to the computing nodes. The second data-placement algorithm is used to reorganize file fragments to solve the data skew problem (see Section 3.2.3). There two cases in which file fragments must be reorganized. In case one, new computing nodes are added to an existing cluster to have the cluster expanded. In case two, new data is appended to an existing input file. In both cases, file fragments distributed by the initial data placement algorithm can be disrupted.

3.2.2 Initial Data Placement

The initial data-placement algorithm begins by dividing a large input file into a number of even-sized fragments. Then, the data placement algorithm assigns fragments to nodes in a cluster in accordance to the nodes' data processing speed. Compared with low-performance nodes, high-performance nodes are expected to store and process more file fragments. Let us consider a Hadoop application processing its input file on a heterogeneous cluster. Regardless of the heterogeneity in node processing power, the initial data placement scheme has to distribute the fragments of the input file in a way that all the nodes can complete processing their local data within almost the same time period.

In our preliminary experiments, we observed that the computing capability of each node in a Hadoop cluster is quite stable for a few tested Hadoop benchmarks, because the response

time of these Hadoop benchmarks on each node is linearly proportional to input data size. As such, we can quantify each node's processing speed in a heterogeneous cluster using a new term called computing ratio. The computing ratio of a computing node with respect to a Hadoop application can be calculated by profiling the application (see Section 3.3.1 for details on how to determine computing ratios). Our preliminary findings show that the computing ratio of a node may vary from application to application.

3.2.3 Data Redistribution

Table 3.1: The Data Redistribution Procedure

Steps	The Data Redistribution Procedures
1	Get the network topology, calculate the computing ratio and utilization
2	Build and sort two lists: under-utilized node list and over-utilized node list
3	Select the source and destination node from the separate lists
4	transfer date from source node to destination node
5	Repeat step 3, 4 until any list is empty

Input file fragments distributed by the initial data-placement algorithm can be disrupted due to one of the following reasons: (1) new data is appended to an existing input file; (2) data blocks are deleted from the existing input file; (3) new data computing nodes are added into an existing cluster, and (4) existing computing nodes are upgraded (e.g., main memory is expanded or hard drives are upgraded to solid state disks). These reasons may trigger the need to solve dynamic data load-balancing problems. To address the dynamic load-balancing issue, we design a data redistribution algorithm to reorganize file fragments based on updated computing ratios.

The data redistribution algorithm 3.1 is described as the following three main steps.

First, like the initial data placement, the data redistribution algorithm must be aware of and collect information regarding the network topology and disk space utilization of a cluster.

Second, the data redistribution algorithm creates and maintains two node lists. The first list contains a set of nodes in which the number of local fragments in each node exceeds its computing capacity. The second list includes nodes that can handle more local fragments thanks to their high performance. The first list is called over-utilized node list; the second list is termed as under-utilized node list.

Third, the data redistribution algorithm repeatedly moves file fragments from an over-utilized node to an underutilized node until data load are evenly distributed and shared among all the nodes. In a process of migrating data between a pair of an over-utilized and an under-utilized nodes, the data redistribution algorithm moves file fragments from a source node in the over-utilized node list to a destination node in the underutilized node list. Note that the algorithm decides the number of bytes rather than fragments and moves fragments from the source to the destination node.

The above load sharing process is repeated until the number of local fragments in each node matches its speed measured by computing ratio. After the data redistribution algorithm is completed, all the heterogeneous nodes in a cluster are expected to finish processing their local data within almost the same time period.

3.3 Implementation of the Data Placement Schemes

3.3.1 Measuring Heterogeneity

Before implementing the initial data placement algorithm, we need to quantify the heterogeneity of a Hadoop cluster in terms of data processing speed. Such processing speed highly depends on data-intensive applications. Thus, heterogeneity measurements in the cluster may change while executing different MapReduce applications. We introduce a metric - called computing ratio - to measure each node's processing speed in a heterogeneous cluster. Computing ratios are determined by a profiling procedure carried out in the following three steps. First, the data processing operations of a given MapReduce application are separately performed in each node. To fairly compare processing speeds, we ensure that all the nodes

process the same amount of data. For example, in one of our experiments the input file size is set to 1GB. Second, we record the response time of each node performing the data processing operations. Third, the shortest response time is used as a reference to normalize the response time measurements. Last, the normalized values, called computing ratios, are employed by the data placement algorithm to allocate input file fragments for the given MapReduce application.

A small computing ratio of a node implies that the node has high speed, indicating that the node should process more file fragments than its slow counterparts.

Now let us make use of an example to demonstrate how to calculate computing ratios that guide the data distribution process. Suppose there are three heterogeneous nodes (i.e., Node A, B and C) in a Hadoop cluster. After running a Hadoop application on each node, we record that the response times of the application on node A, B and C are 10, 20 and 30 seconds, respectively. The response time of the application on node C is the shortest. Therefore, the computing ratio of node A with respect to this application is set to 1, which becomes a reference used to determine computing ratios of node B and C. Thus, the computing ratios of node B and C are 2 and 3, respectively. Recall that the computing capacity of each node is quite stable with respect to a Hadoop application. Hence, the computing ratios are independent of input file sizes. Now, the least common multiple of these ratios 1, 2, 3 is 6. We divide 6 by the ratio of each node to get its portion. Table 3.2 shows the response times and computing ratios for each node in a Hadoop cluster. Table 3.2 shows the number of file fragments to be distributed to each node in the cluster. Intuitively, the fast computing node (i.e., node A) has to handle 60 file fragments whereas the slow node (i.e., 3) only needs to process 20 fragments.

3.3.2 Sharing Files among Multiple Applications

The heterogeneity measurement of a cluster depends on data-intensive applications. If multiple MapReduce applications must process the same input file, the data placement

Table 3.2: Computing ratios, response times, and number of file fragments for three nodes in a Hadoop cluster

Node	Response time	Ratio	File fragments	Speed
Node A	10	1	6	Fastest
Node B	20	2	3	Average
Node C	30	3	2	Slowest

mechanism may need to distribute the input file's fragments in several ways - one for each MapReduce application. In the case where multiple applications are similar in terms of data processing speed, one data placement decision may fit the needs of all the applications.

3.3.3 Data Distribution.

File fragment distribution is governed by a data distribution server, which constructs a network topology and calculates disk space utilization. For each MapReduce application, the server generates and maintains a configuration file containing a list of computing-ratio information. The data distribution server applies the round-robin policy to assign input file fragments to heterogeneous nodes based on their computing ratios. When a new Hadoop application is installed on a cluster, the application's configuration file will be created by the data distribution server. In case any node of a cluster or the entire cluster is upgraded, the configuration files of all the Hadoop applications installed in the cluster must be updated by the data distribution server. This update process is important because computing ratios are changing after any update on the cluster.

3.4 Performance Evaluation

In this part of the study, we use two data-intensive applications - Grep and WordCount - to evaluate the performance of our data placement mechanism in a heterogeneous Hadoop cluster. The tested cluster consists of five heterogeneous computing nodes (see Table 3.3 for the configuration summary of the cluster). Both Grep and WordCount are two Hadoop applications running on the tested cluster. Grep is a tool searching for a regular expression

in a text file; whereas WordCount is a program used to count the number of words in text files.

Table 3.3: Five Nodes in a Hadoop Heterogeneous Cluster

Node	CPU Model	CPU(hz)	L1 Cache(KB)
Node A	Intel Core 2 Duo	$2 \times 1G = 2G$	204
Node B	Intel Celeron	2.8G	256
Node C	Intel Pentium 3	1.2G	256
Node D	Intel Pentium 3	1.2G	256
Node E	Intel Pentium 3	1.2G	256

The data distribution server follows the approach described in Section 3.3.1 to obtain computing ratios of the five computing nodes with respect to the Grep and WordCount applications (see Table 3.4). The computing ratios shown in Table 3.4 represent the heterogeneity of the Hadoop cluster with respect to Grep and WordCount. The information contained in Table 3.4 is created by the data distribution server and is stored in a configuration file by this server.

We observe from Table 3.4) that computing ratios of a Hadoop cluster are application dependent. For example, node A is 3.3 times faster than nodes C-E with respect to the Grep application; node A is 5 (rather than 3.3) times faster than nodes C-E when it comes to the WordCount application. The implication of the results is that given a heterogeneous cluster, one has to determine computing ratios for each Hadoop application. Note that computing ratios of each application only needs to be calculated once for each cluster. If any hardware component of a cluster is updated, computing ratios stored in the configuration file must be determined by the data distribution server again.

Figures 3.1 and 3.2 show the response times of the Grep and WordCount applications running on each node of the Hadoop cluster when the input file size is 1.3 GB and 2.6 GB, respectively. The results plotted in Figures 3.1 and 3.2 suggest that computing ratios are independent of input file size, because the response times of Grep and WordCount are proportional to the file size. Regardless of input file size, the computing ratios for Grep and WordCount on the 5-node Hadoop clusters remain unchanged (see Table 3.4 for the ratios).

Figure 3.1: Response time of Grep on each node

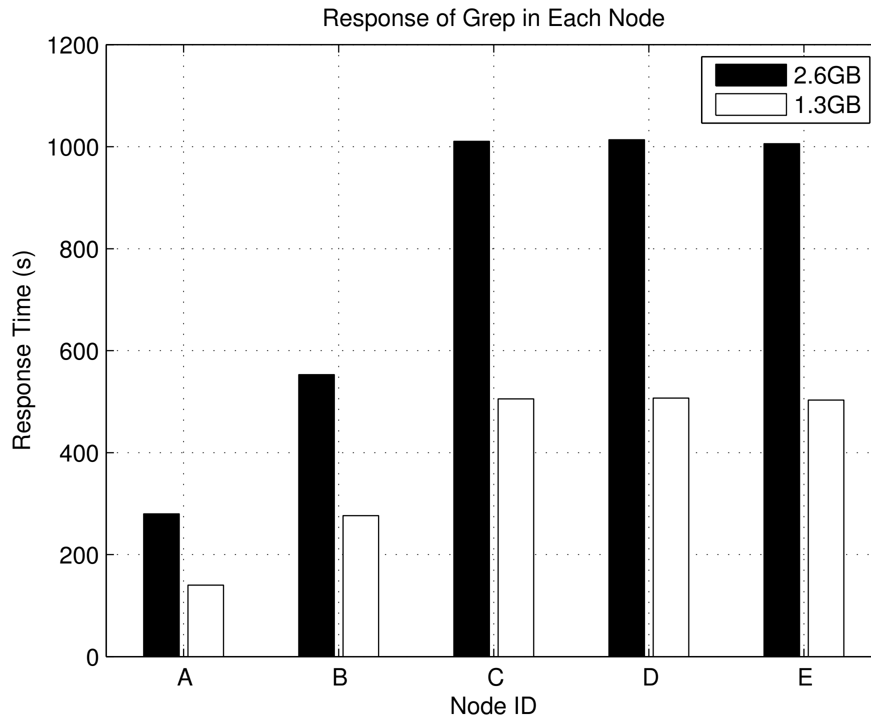


Figure 3.2: Response time of Wordcount on each node

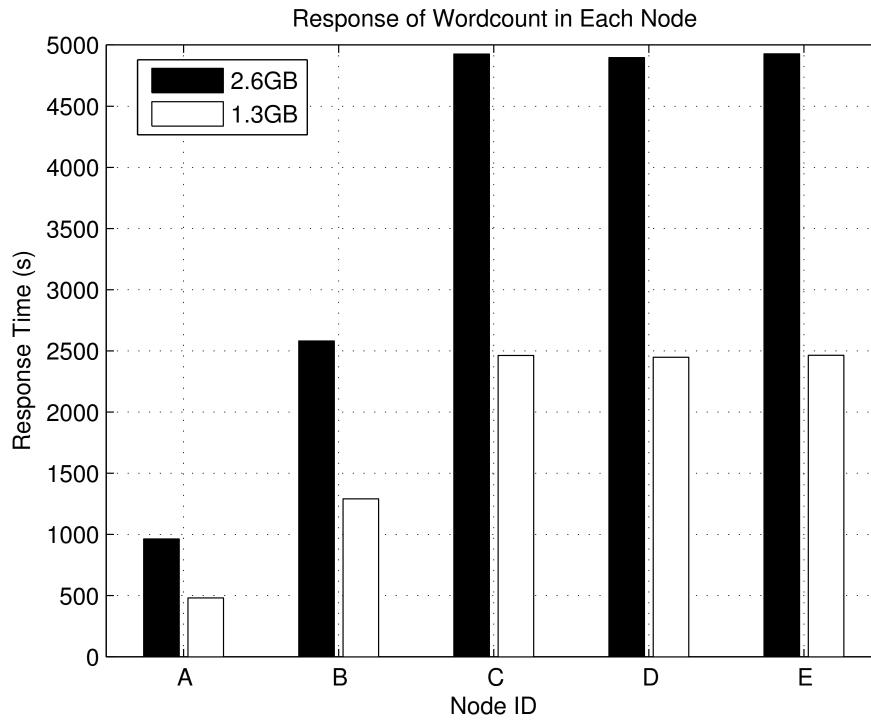


Figure 3.3: Impact of data placement on performance of Grep

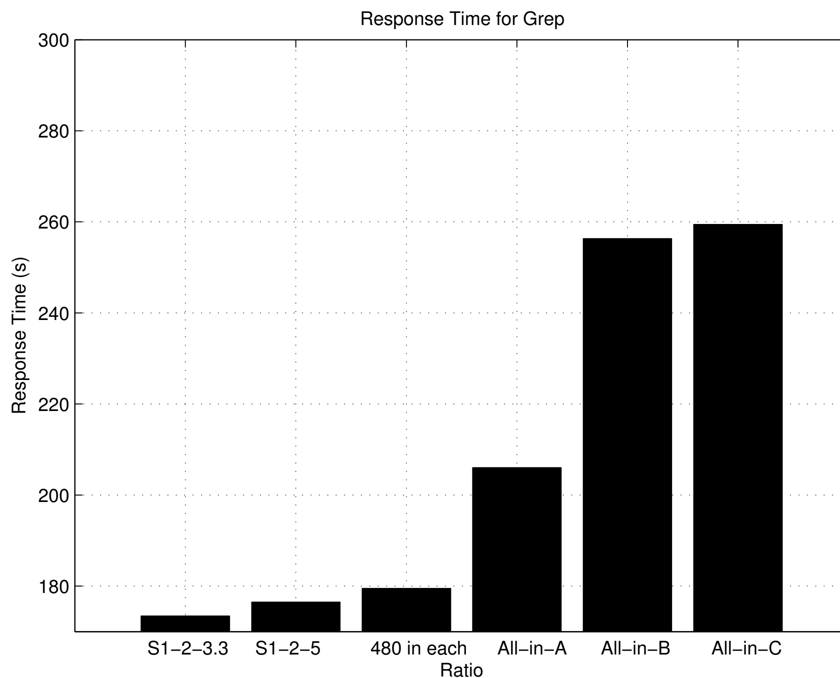


Figure 3.4: Impact of data placement on performance of WordCount

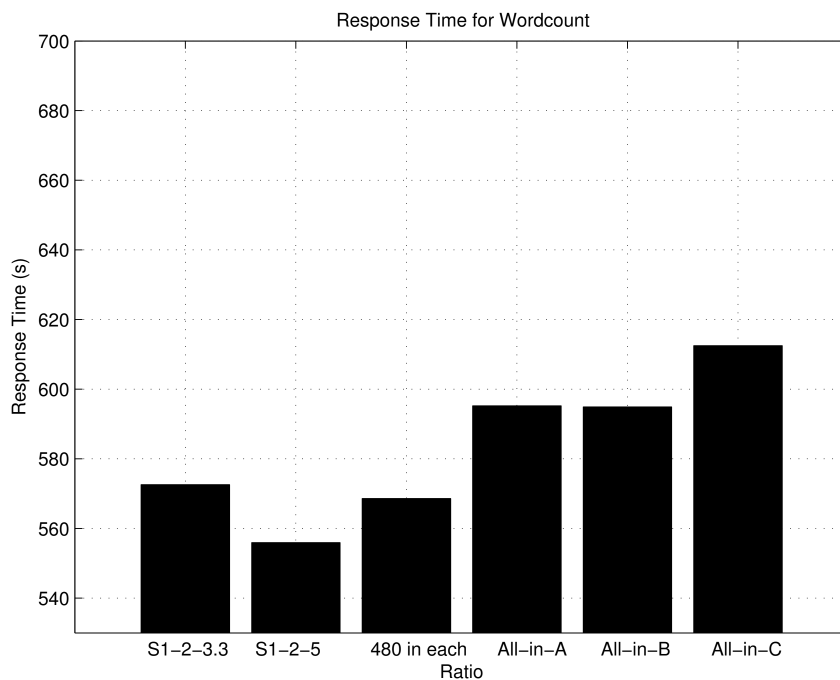


Table 3.4: Computing Ratios of the Five Nodes with Respective of the Grep and WordCount Applications

Computer Node	Ratios for Grep	Ratios for WordCount
Node A	1	1
Node B	2	2
Node C	3.3	5
Node D	3.3	5
Node E	3.3	5

Given the same input file size, Grep's response times are shorter than those of WordCount (see Figs. 3.1 and 3.2). As a result, the computing ratios of Grep are different from those of WordCount (see Table 3.4).

Table 3.5: Six Data Placement Decisions

Notation	Data Placement Decisions
S1-2-3.3	Distributing files under the computing ratios of the grep. (This is an optimal data placement for Grep)
S1-2-5	Distributing files under the computing ratios of the wordcount. (This is an optimal data placement for WordCount)
480 in each	Average distribution of files to each node.
All-in-A	Allocating all the files to node A.
All-in-B	Allocating all the files to node B.
All-in-C	Allocating all the files to node C.

Now we are positioned to evaluate the impacts of data placement decisions on the response times of Grep and WordCount (see Figures 3.3 and 3.4). Table 3.5 shows six representative data placement decisions, including two optimal data-placement decisions (see S1-2-3.3 and S1-2-5 in Table 3.5) offered by the data placement algorithm for the Grep and WordCount applications. The file fragments of input data are distributed and placed on the five heterogeneous nodes based on six different data placement decisions, among which two optimal decisions (i.e., S1-2-3.3 and S1-2-5 in Table 3.5) are made by our data placement scheme based on the computing ratios stored in the configuration file (see Table 3.4).

Let us use an example to show how the data distribution server relies on the S1-2-3.3 decision - optimal decision for Grep - in Table 3.5 to distribute data to the five nodes of the

tested cluster. In accordance with the configuration file managed by the data distribution server, the computing ratios of Grep on the 5-node Hadoop cluster are 1, 2, 3.3, 3.3, and 3.3 for nodes A-E (see Table 3.4). We suppose there are 24 fragments of the input file for Grep. Thus, the data distribution server allocates 10 fragments to node A, 5 fragments to node B, and 3 fragments to nodes C-E.

Figure 3.3 reveals the impacts of data placement on the response times of the Grep application. The first (leftmost) bar in Figure 3.3 shows the response time of the Grep application after distributing file fragments based on Grep's computing ratios. For comparison purpose, the other bars in Figure 3.3 show the response time of Grep on the 5-node cluster with the other five data-placement decisions. For example, the third bar in Figure 3.3 is the response time of Grep when all the input file fragments are evenly distributed across the five nodes in the cluster.

We observe from Figure 3.3 that the first data placement decision (denoted as S1-2-3.3) leads to the best performance of Grep, because the input file fragments are distributed strictly according to the nodes' computing ratios. If the file fragments are placed using the "All-in-C" data-placement decision, Grep performs extremely poorly. Grep's response time is unacceptably long under the "All-in-C" decision, because all the input file fragments are placed on node C - one of the slowest node in the cluster. Under the "All-in-C" data placement decision, the fast nodes (i.e., nodes A and B) have to pay extra overhead to copy a significant amount of data from node C before locally processing the input data. Compared with the "All-in-C" decision, the optimal data placement decision reduces the response time of Grep by more than 33.1%.

Figure 3.4 depicts the impacts of data placement decisions on the response times of the WordCount application. The second bar in Figure 3.4 demonstrates the response time of the WordCount application on the cluster under an optimal data placement decision. In this optimal data placement case, the input file fragments are distributed according to the computing ratios (see Table 3.4) decided and managed by the data distribution server. To

illustrate performance improvement achieved by our new data placement strategy, we plotted the other five bars in Figure 3.4 to show the response time of WordCount when the other five data-placement decisions are made and applied. The results plotted in Figure 3.4 indicate that the response time of WordCount under the optimal "S1-2-5" data placement decision is the shortest compared with all the other five data placement decisions. For example, compared with the "All-in-C" decision, the optimal decision made by our strategy reduces the response time of WordCount by 10.2%. The "S1-2-5" data placement decision is proved to be the best, because this data placement decision is made based on the heterogeneity measurements - computing ratios in Table 3.4. Again, the "All-in-C" data placement decision leads to the worst performance of WordCount, because under the "All-in-C" decision the fast nodes have copy a significant amount of data from node C. Moving data from node C to other fast nodes introduces extra overhead.

In summary, the results reported in Figures 3.3 and 3.4 show that our data placement scheme can improve the performance of Grep and Wordcount by up to 33.1% and 10.2% with averages of 17.3% and 7.1%, respectively.

3.5 Summary

In this Chapter, we described a performance problem in HDFS (Hadoop Distributed File System) on heterogeneous clusters. Motivated by the performance degradation caused by heterogeneity, we designed and implemented a data placement mechanism in HDFS. The new mechanism distributes fragments of an input file to heterogeneous nodes according to their computing capacities. Our approach significantly improves performance of Hadoop heterogeneous clusters. For example, the empirical results show that our data placement mechanism can boost the performance of the two Hadoop applications (i.e., Grep and WordCount) by up to 33.1% and 10.2% with averages of 17.3% and 7.1%, respectively.

In a future study, we will extend this data placement scheme by considering the data redundancy issue in Hadoop clusters. We also will design a dynamic data distribution

mechanism for mutliple data-intensive applications sharing and processing the same data sets.

Chapter 4

Predictive Scheduling and Prefetching for Hadoop clusters

In Chapter 2.1, we introduced MapReduce - a programming model and framework that has been employed to develop a wide variety of data-intensive applications in large-scale systems. Recall that Hadoop is a Yahoo's implementation of the MapReduce model. In the previous Chapter, we proposed a novel data placement scheme to improve performance of heterogeneous Hadoop clusters. In this Chapter, we focus on predictive scheduling and prefetching issues in Hadoop clusters.

4.1 Motivations for a New Prefetching/Scheduling Mechanism in Hadoop

4.1.1 Data Locality Problems in Hadoop

In this Chapter, we first observe the data movement and task process patterns of Hadoop. Then, we identify a data locality problem in Hadoop. Next, we design a predictive and scheduling mechanism called PSP to solve the data locality problem to improve the performance of Hadoop. We show a way of aggressively searching for subsequent blocks to be prefetched, thereby avoiding I/O stalls incurred by data accesses. At the core of our approach is a predictive scheduling module, which can be integrated with the native Hadoop system.

In what follows, we highlight four factors making predictive scheduling and prefetching very desirable and possible:

1. the underutilization of CPU processes in data nodes of a Hadoop cluster;

2. the growing importance of Hadoop performance;

3. the data storage information offered by the Hadoop distribution file system (HDFS); and
4. interaction between the master node ant slave nodes (a.k.a., data nodes).

Our preliminary results show that CPU and I/O workload are underutilized when a data-intensive application is running on a Hadoop cluster. In Hadoop, HDFS is tuned to support large files and; typically, file sizes are ranging from gigabytes to terabytes. HDFS (see Chapter 2.2 for details on HDFS) spilts a large file to several partitions and distributes to multiple nodes in a Hadoop cluster. HDFS handles the index information - called meta data - of large files to manage their file partitions. These partitions are the the basic data elements in HDFS; the size of the partitions by default is 64 MB. Please note that the big block size (i.e., 64 BM) can shorten disk seeking times; however, because of the large block size, the data transfer time dominates the entire I/O access time of the large blocks. In addition to large data transfer times, and I/O stalls are also a significant factor in the data processing times. This noticeable I/O stalls motivate us to investigate prefetching techniques to boost I/O performance of HDFS and improve the performance of Hadoop clusters.

The second factor encouraging us to study the prefetching issue in Hadoop is that high-performance CPUs are processing data much faster than disks can read and write data. Simply increasing I/O caches can not continue improving the performance of I/O systems and CPUs [51]. In Hadoop clusters, before a computing node launches a new task, the node requests task assignments from the master node in the clusters. The master node informs the computing node important meta data, which includes not only the next task to be running on the node but also the location of the data to be processed by the task. The computing node does not retrieve required input data until the data's meta-data become available. This procedure implies that the CPU of the computing node has to wait for a noticeable time period while the node is communicating with the master node to acquire the meta-data. We believe that a prefetching scheme can be incorporated into this data processing procedure in Hadoop to prevent CPUs from waiting for the master node to deliver meta-data.

A master node (a.k.a., NameNode) in HDFS manages meta data of input files, whereas input data sets are stored in slave nodes (a.k.a., DataNodes). This characteristic of NameNode allows us to access each block in a large file through the file's meta-data. Hadoop applications like web-index and search engines are data-intensive in general and read-intensive in particular. The access patterns of Hadoop applications can be tracked and predicted for the purpose of data accessing and task scheduling in Hadoop clusters.

In this Chapter, we present a predictive scheduling and prefetching mechanism that aims at improving the performance of Hadoop clusters. In particular, we propose a predictive scheduling algorithm to assign tasks to DataNodes in a Hadoop cluster. The prefetching scheme described in this Chapter manages the data loading procedure in HDFS. The basic idea of our scheduling and prefetching mechanism is to preload input data from local disks and place the data into the local cache of the DataNodes as late as possible without any starting delays of new tasks assigned to the DataNodes.

The novelty of this part of the dissertation study lies in our new mechanism that integrates a prefetching scheme with a predictive scheduling algorithm. The original Hadoop system randomly assigns tasks to computing nodes and loads data from local or remote disks whenever the data sets are required. CPUs of the computing nodes will not process new tasks until all the input data resources are loaded into the nodes' main memory. The coordination between CPUs and disks in terms of data I/O has a negative impact on Hadoop's performance. In the design of our mechanism, we change the order of the processing procedure, our prefetching scheme assists Hadoop clusters to preload required input data prior to launching tasks on DataNodes.

4.1.2 Contributions of our Prefetching and Scheduling Mechanism

The major contribution of this Chapter is a prefetching algorithm and a predictive scheduling algorithm. The integration of the two algorithms aim at the following four goals:

1. to preload input data from local disks prior to new task assignments;

2. to shorten CPU waiting times of DataNodes;

3. to start running a new task immediately after the task is assigned to a DataNode; and

4. to improve the overall performance of Hadoop clusters.

We evaluate our prefetching and scheduling solutions using a set of Hadoop benchmarks on a real-world cluster. Evaluation results show that our prefetching and scheduling mechanism can achieve at least 10% reduction in execution times compared with the native Hadoop system.

4.1.3 Chapter Organization

The rest of this Chapter is organized as follows. Section 4.2 first describes the system architecture followed by the design of prefetching and scheduling algorithms. Section **??** highlights the implementation details of our prefetching and scheduling mechanism. In Section 4.3, we present the evaluation results and Section 4.4 summarizes this Chapter.

4.2 Design and Implementation Issues

In this section, we present the challenges and goals on designing our prefetching and scheduling mechanism in the context of Hadoop clusters. Then, we discuss the components of this mechanism in detail.

4.2.1 Desgin Challenges

A variety of scheduling technologies are now available; it is likely to address the performance problem described in the previous section from computation perspective. Such scheduling methods arrange tasks and sequences to each computing node of a cluster. However, the problem always exist that huge mount data should be loaded to main memory before tasks are launched on the nodes. The goal of this study in our dissertation research is to

investigate scheduling and prefetching methods for successfully reducing perceived latencies associated with the HDFS file system operations.

One of Hadoop's design principles is that moving computation is cheaper than moving data. This principle indicates that it is often efficient to migrate processing tasks closer to where input data is located rather than moving data toward to a node where tasks are running. This principle is especially true when the size of data sets is huge, because the migration of computations minimizes network congestions and increases the overall throughput of Hadoop clusters. A recent study [64] shows that the best case of task scheduling in HDFS is when the scheduler assigns corresponding tasks into the local node. The second best case is when the scheduler assigns tasks into the local rack.

Most of the existing scheduling algorithms focus on improving the performance of CPUs. In addition to CPU performance, data locality is another important issue to be addressed in clusters. In our previous Chapter, we described our new data placement algorithm applied to distribute input data according to DataNodes' computing capability. In our data placement scheme, fast nodes are assigned more data than slow ones. A data-locality-aware scheduling mechanism can directly allocate more tasks to fast nodes than slow nodes.

Some characteristics of the Hadoop system make data prefetching in Hadoop's file system quite different from prefetching in other files systems. In what follows, we present three challenges involved in building our prefetching mechanism for the Hadoop system. The main idea of our design is to preload input data within a single block while performing a CPU-intensive task on a DataNode. When a map task is running on a DataNode, the to-be-required data is prefetched and stored in the cache of the DataNode. In order to preload data prior to task assignments, we need to consider the following issues:

1. Which data blocks should be preloaded?

2. Where are data blocks located?

3. How to synchronize computing tasks with the data prefetching process?

4. How to optimize the size of cache for prefetched data.

The first two issues in the above list deal with what data blocks to be prefetched. The third issue in the list is focused on the best time point to trigger the prefetching procedure. For example, if data blocks are fetched into cache too earlier, the scarce cache in DataNodes is underutilized. In contrast, if the data blocks are fetched too late, CPU waiting times are increased. The last issue in the list is related to a way of efficiently prefetching data blocks in HDFS. For example, we must determine the best size of prefetched data in each DataNode to fully utilize the cache resources. If the prefetched data size is optimized, then our prefetching mechanism can maximize benefit for Hadoop clusters by minimizing the prefetching overhead.

4.2.2 Objectives

The goal of this study is to investigate methods for reducing data accessing times by hiding I/O latencies in Hadoop clusters. There are the following three objectives in this part of the study:

1. We propose a data-locality aware scheduling mechanism. We examine the feasibility of improving the performance of Hadoop by hiding I/O accessing latencies.

2. We develop a prefetching scheme to boost I/O performance of HDFS.

3. To quantify the benefits of our prefetching strategy, we compare the response time of benchmarks running on a Hadoop cluster equipped with our prefetching mechanism against the same cluster without adopting our scheme.

4.2.3 Architecture

Recall that Hadoop is a Yahoo's open-source implementation of the MapReduce programming model [78]. Hadoop is widely deployed in large-scale clusters in data centers in many companies like Facebook, Amazon, and the New York Times. Hadoop relies on its

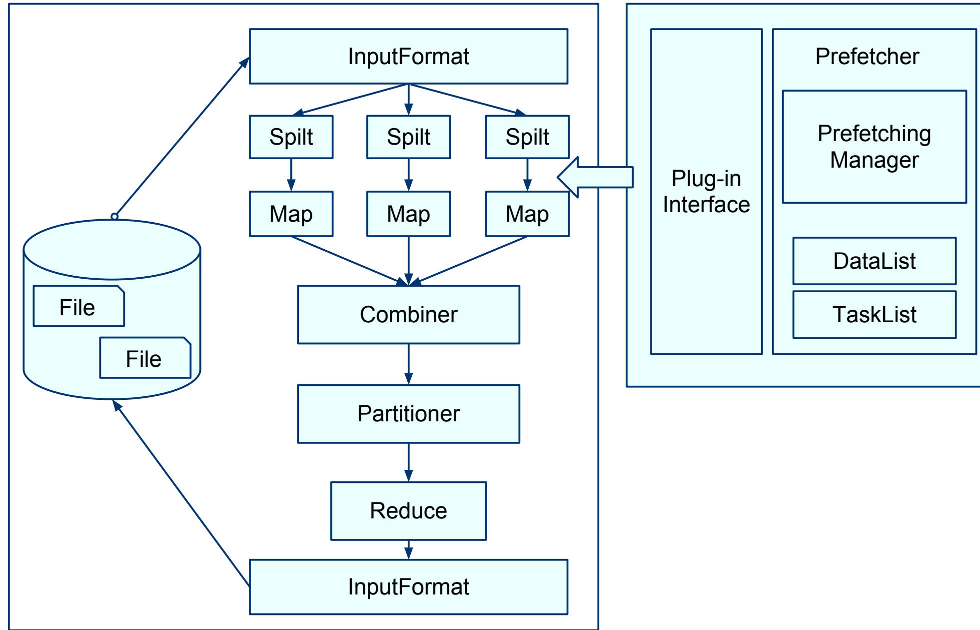


Figure 4.1: The architecture and workflow of MapReduce

distributed file system called HDFS (Hadoop Distributed File System) [13] to manage a massive amount of data. The Hadoop running system coupled with HDFS manages the details parallelism and concurrency to provide ease of parallel programming with reinforced reliability. Moreover, Hadoop is a java software framework that supports data-intensive distributed applications [23]. Please refer to Chapter 2.2 for more background information on Hadoop and HDFS.

Figure 4.1 illustrates the general architecture and the typical workflow of the Hadoop system. An input file is partitioned into a set of blocks (a.k.a., fragments) distributed among DataNodes in HDFS. Map tasks process these small data block and generate intermediate outputs. Multiple intermediate outputs generated from the DataNoodes are combined into to a single large intermediate output. The partitioner controls $< key, value >$ pairs of the intermediate map results. Therefore, the $< key, value >$ pairs with the same key are shuffled to the same reduce task to be further sorted and processed.

In the above procedure, huge amounts of data are loaded from disk to main memory. Nevertheless, our preliminary experiments indicate that the bandwidths of disks in DataNodes of HDFS are not saturated. The preliminary findings suggest that the underutilized disk bandwidth during the above shuffling process can be leveraged to prefetch data blocks.

4.2.4 Predictive Scheduler

We design a predictive scheduler - a flexible task scheduler - to predict the most appropriate task trackers to which future tasks should be assigned. Once the scheduling decisions are predicted ahead of time, DataNodes can immediately start loading $< key, value >$ pairs. Our predictive scheduler allows DataNodes to explore the underutilized disk bandwidth by preloading $< key, value >$ pairs.

Let us start describing this scheudling mechanism by introducing the native Hadoop scheduler. The job tracker includes a task scheduler module to assign tasks to different task trackers. The task tracker periodically sends a heartbeat to the job tracker. The job tracker checks heartbeat and assigns tasks to available task trackers. The scheduler assigns each task to a node randomly via the same heartbeat message protocol. The algorithm for predicting stragglers in the native Hadoop is inadequate, because the original algorithm uses a single heuristic variable for prediction purpose. The native Hadoop randomly assigns tasks and mispredicts stragglers in many cases.

To address the aforementioned problem, we develop a predictive scheduler by designing a prediction algorithm integrated with the native Haddop. Our predictive scheduler seeks stragglers and predicts candidate data blocks. The prediction results on the expected data are sent to corresponding tasks. The prediction decisions are made by a prediction module during the prefetching stage.

We seamlessly integrate the predictive scheduler with the prefetching module. Below let us describe the structure of the prefetching module, which consists of a single prefetching manager and multiple worker threads. The role of the prefetching manager is to monitor

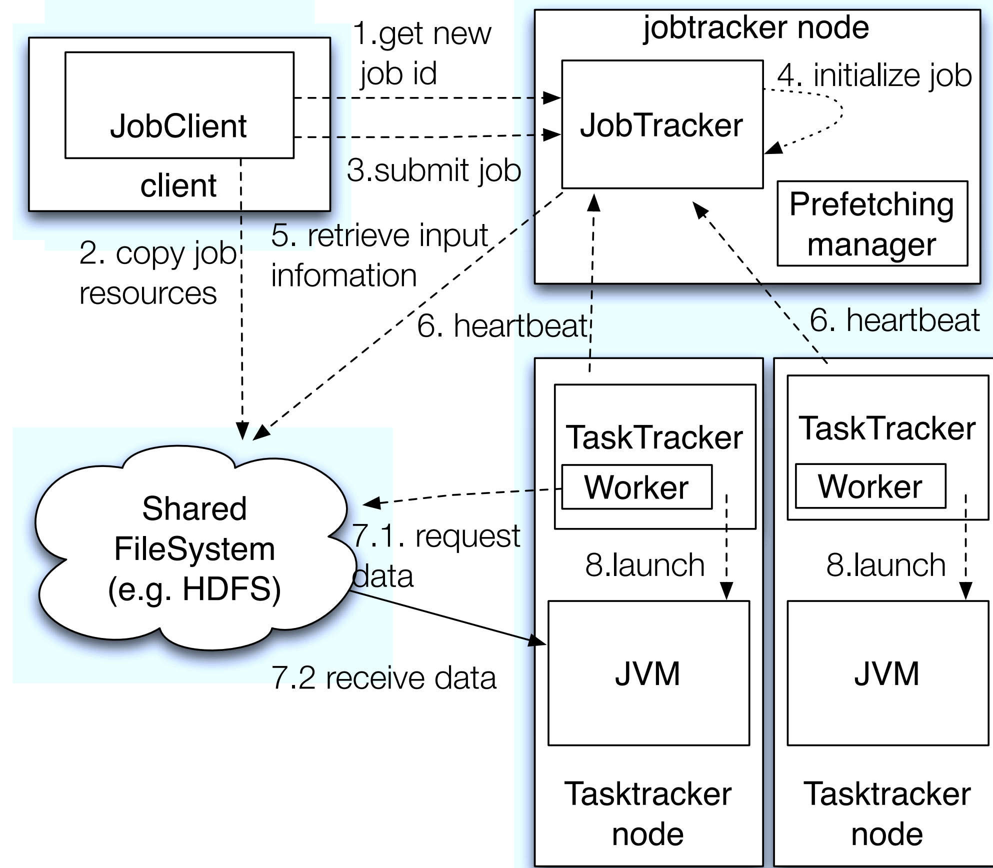


Figure 4.2: Three basic steps to launch a task in Hadoop.

the status of worker threads and to coordinate the prefetching process with tasks to be scheduled. When the job tracker receives a job request from a Hadoop application, the job tracker places the job in an internal queue and initializes the job [76][73]. The job tracker divides a large input file to several fixed-size blocks and creates one map task for each block. Thus, the job tracker partitions the job into multiple tasks to be processed by task trackers. When the job tracker receives a heartbeat message from an idle task tracker, the job tracker retrieves a task from the queue and assigns the task to the idle task tracker. After the task tracker obtains the task from the job tracker, the task is running on the task tracker.

Figure 4.2 shows that the Hadoop system applies the following three basic steps to launch a task. First, the job tracker localizes the job JAR by copying the job from the shared file system to the task tracker's file system. The job tracker also copies any required

files by the Hadoop application from the distributed cache to the local disk. Second, the job tracker creates a local working directory for the task, and un-jars the contents of the JAR into this directory. Last, an instance of TaskRunner is created to launch a new Java Virtual Machine to run the task.

In our design, the above task launching procedure is monitored by the prediction module. Specifically, the prediction module in the scheduler predicts the following events.

1. finish times of tasks currently running on nodes of a Hadoop cluster;
2. pending tasks to be assigned to task trackers; and
3. launch times of the pending tasks.

4.2.5 Prefetching

Upon the arrival of a request from the Job tracker, the predictive scheduler triggers the prefetching module that forces preload worker threads to start loading data to main memory. The following three issues must be addressed in the prefetching module.

When to prefetch. In this first issue, the prefetching module controls how early to trigger prefetching actions. In the previous Chapter, we showed that one node process the same block size data in a fix time period. Before a block finishes, the subsequent block will be loaded into the main memory of the node. The prediction module assists the prefetching module to estimate the execution time of processing each block in a node. Please note that the block processing time of an application on different nodes may vary in a heterogeneous cluster. The estimates are calculated by statistically measuring the processing times of blocks on all the nodes in a cluster. This statistic measuring can be performed offline.

What to prefech. In the second issue, the prefetching module must determine blocks to be prefetched. Initially, the predictive scheduler assigns two tasks to each task tracker in a node. When the prefetching module is triggered, it proactively contacts the job tracker to seek required information regarding data to be processed by subsequent tasks.

How much to prefetch. In the last issue, the prefetching module decides the amount of data to be preloaded. When one task is running, the predictive scheduler manages one or more waiting tasks in the queue. When the prefetching action is triggered, the prefetching module automatically fetches data form disks. Due to the large block size in HDFS, we intend not to make our prefetching module very aggressive. Thus, there is only one block being prefeched at a time.

The most important part of the prefetching work is to synchronize two resources in the MapReduce system: the computing task and the data block. The scheduler in the MapReduce always collects all the running task information and constructs a RunningTaskList. It separately caches the different types of tasks in a map task list and a reduce task list. The job tracker can manage the current task according to these lists [64]. The prefetching manager in the master node constructs a list known as the data list, a collection of all the data block location information.

The role of worker thread in each node is to load the file into the memory. In the native MapReduce system, this step is processed in the initial function (localizejob) after the task tracker receives the task command. In our design, the prefetching manager provides the block location and task environment information to a worker thread. The worker thread can finish the data loading job all by itself before the task is received.

4.3 Performance Evaluation

4.3.1 Experimental Environment

To evaluate the performance of the proposed predictive scheduling and prefetching mechanism, we run Hadoop benchmarks on a 10-node cluster. Table 4.1 summarizes the configuration of the cluster used as a testbed of the performance evaluation. Each computing node in the cluster is equipped with two dual-core 2.4 GHz Intel processors, 2GB main memory, 120GB SATA hard disk, and a Gigabit Ethernet network interface card.

In our experiments, we configure the block size of HDFS to be 64 MB and the number of replicas of each data block to be one. It does not imply by any means that one should not increase the number of replica to three - a default value in HDFS. In this study, we focus on impact of predictive scheduling and prefetching on Hadoop. We intentionally disable the data replica feature of HDFS, because data replicas make performance impacts on HDFS. Our predictive scheduling and prefetching mechanism can be employed in combination with the data replica mechanism to further improve performance of HDFS.

Table 4.1: Test Setting

CPU	Intel Xeon 2.4GHz
Memory	2GB Memory
Disk	SEGATE 146GB
Operation System	Ubuntu 10.4
Hadoop version	0.20.2

In our experiments, we test the following two Hadoop benchmarks running on the Hadoop system, in which the predictive scheduler is incorporated to improve the performance of the Hadoop cluster.

1. WordCount (WC): WordCount counts the frequency of occurrence for each word in a text file. The Map tasks process different sections of input files and return intermediate data that consists of several pairs word and frequency. Then, the Reduce tasks add up the values for each identity word. The Word-Count is considered as a memory-intensive application.

2. Grep (GR): Grep is a searching tool for a regular expression in a text file. Unlike WordCount, Grep is a data-intensive application.

4.3.2 Individual Node Evaluation

Figure 4.3 shows the execution times of the Grep application running in our prefetching-enabled Hadoop system (PSP) and the native Hadoop system. To demonstrate the impact

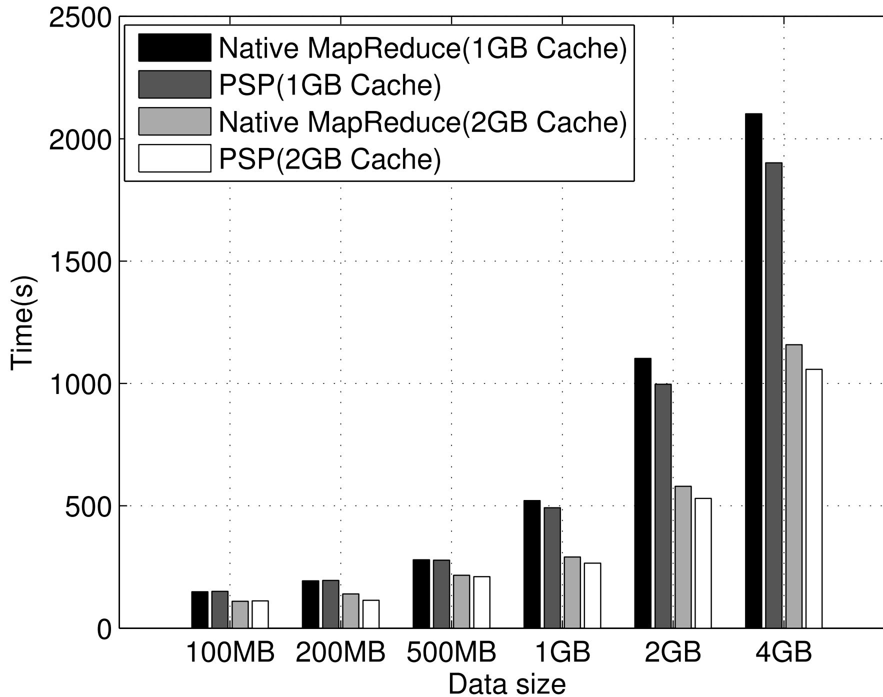


Figure 4.3: The execution times of Grep in the native Hadoop system and the prefetching-enabled Hadoop system (PSP).

of main memory on the performance of our prefetching-enabled Hadoop, we choose the main memory size of the computing nodes in our cluster to be 1 GB to 2 GB. The first two bars in each result group in Figure 4.3 are the response times of Grep on the cluster in which each node has 1 GB of memory. The last two bars in each result group in Figure 4.3 are the response times of Grep on the same cluster in which each node has 2 GB of memory.

Figure 4.3 shows that performance of the Hadoop cluster is very sensitive to main memory size. For example, when the input data size is 100MB, increasing the memory size from 1 GB to 2 GB can reduce the execution time of the Grep benchmark by 32%. When the input data size becomes 4GB, a large memory capacity (i.e., 2 GB) reduces Grep's execution time by 45% compared with the same cluster with small memory capacity (i.e., 1 GB). Our results indicate that a larger input file makes the Hadoop cluster more sensitive to the memory size. Intuitively, increasing memory size is an efficient way to boost the performance

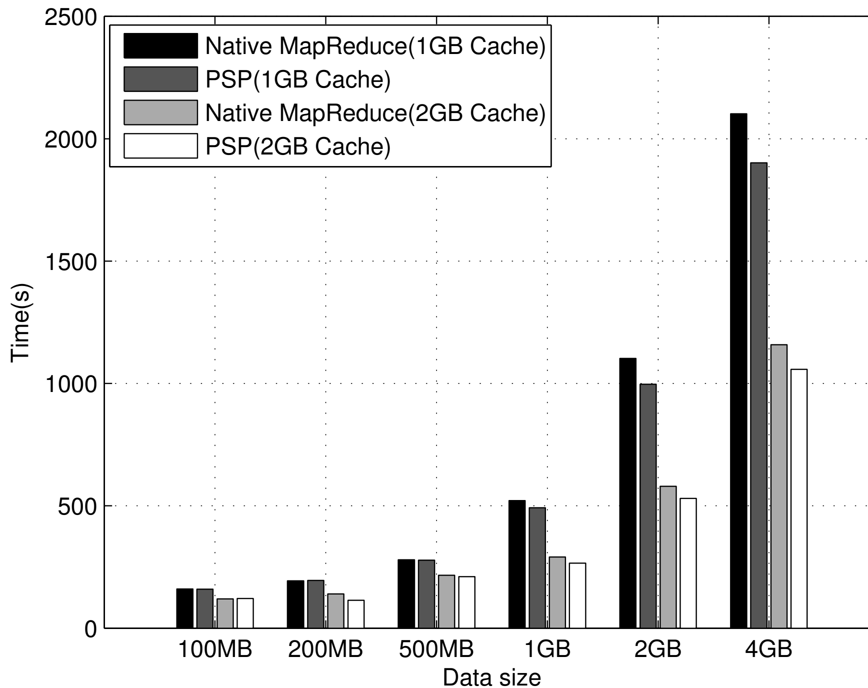


Figure 4.4: The execution times of WordCount in the native Hadoop system and the prefetching-enabled Hadoop system (PSP).

of Hadoop cluster processing large files. It is worth noting that expanding memory size of Hadoop cluster is not a cost-effective way of boosting system performance.

Figure 4.3 also reveals that when the input data size is smaller than or equal to 500MB, our predictive scheduling and prefetching module does not make any noticeable impact on the performance of Hadoop. In contrast, when it comes to large input data size (e.g., 2 GB and 4 GB), the predictive scheduling and prefetching (PSP) significantly reduces the response time of Grep by 9.5% (for the case of 1 GB memory) and 8.5% (for the case of 2 GB memory), respectively.

Figure 4.4 shows the execution times of the WordCount benchmark running in both our prefetching-enabled Hadoop system (PSP) and the native Hadoop system. The performance trend illustrated in Figure 4.4 is very similar to that observed in Figure 4.3. For example, Figure 4.4 suggests that memory size has significant impacts on the execution time of the WordCount benchmark on Hadoop clusters when the input file is large. The experimental

Figure 4.5: The performance of Grep and WordCount when a single large file is processed by the prefetching-enabled Hadoop system (PSP).

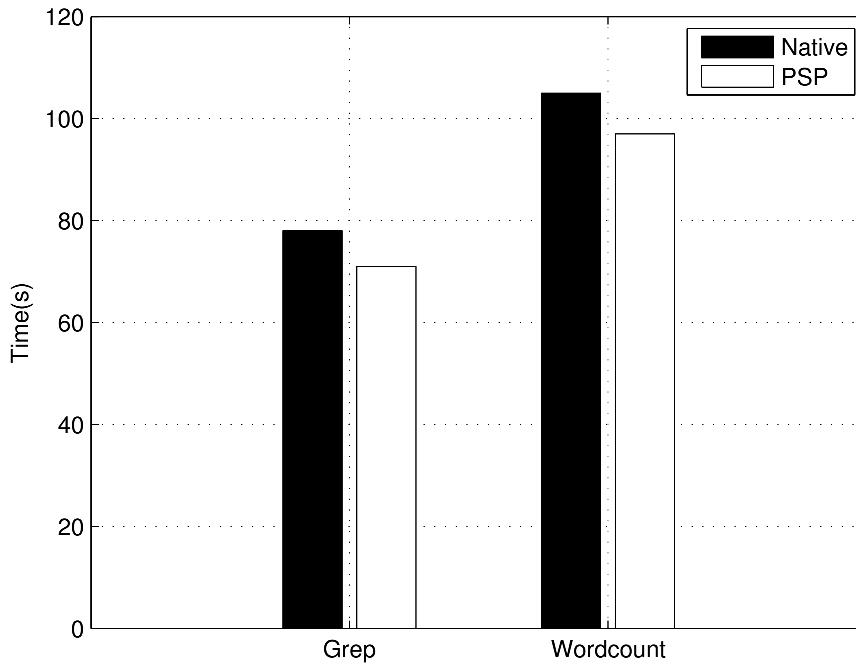
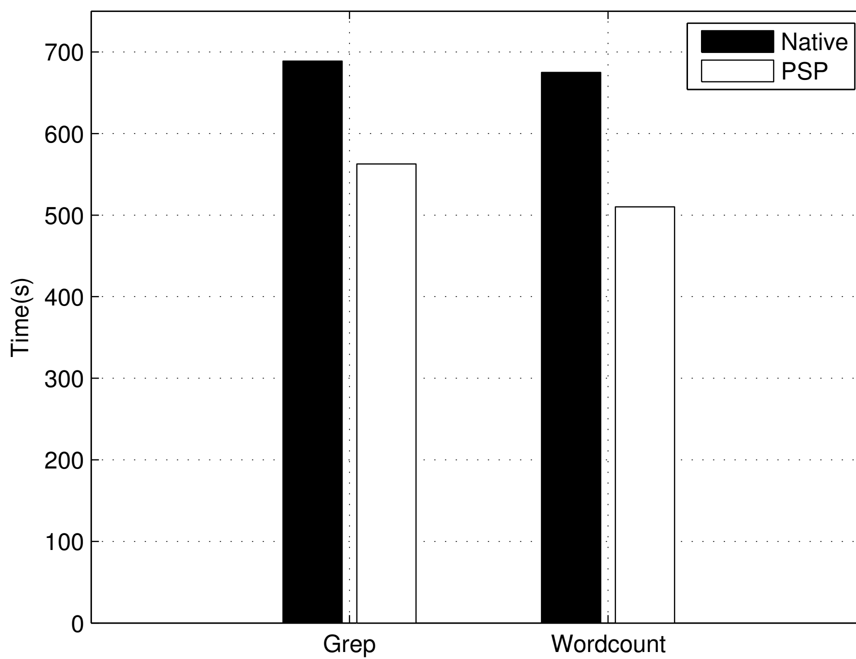
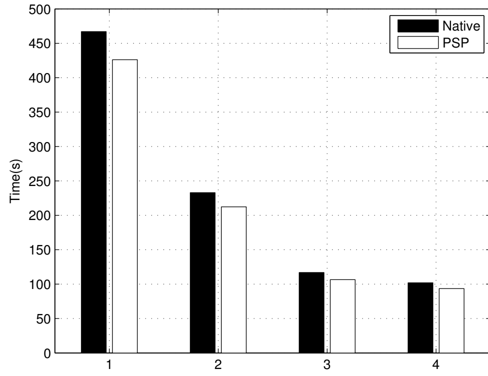
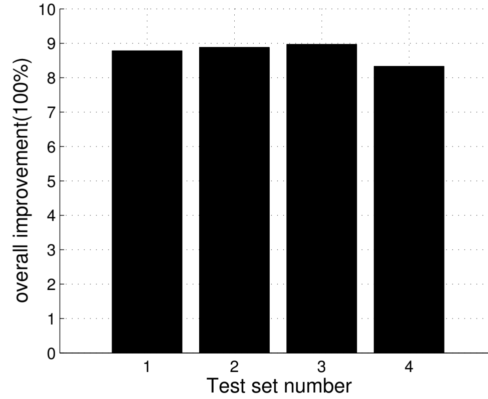


Figure 4.6: The performance of Grep and WordCount when multiple small files are processed by the prefetching-enabled Hadoop system (PSP).

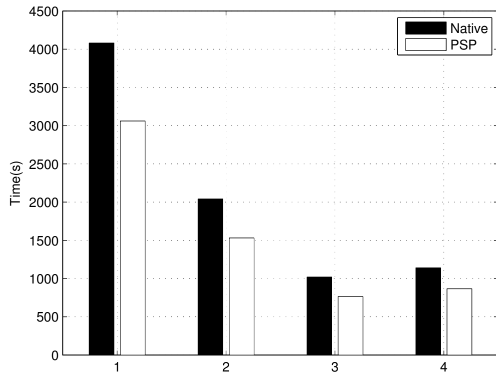




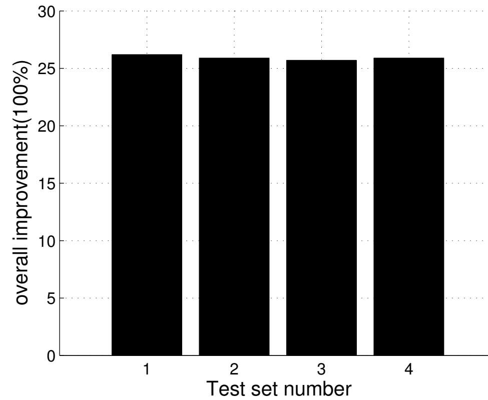
(a) The response time comparison between the native Hadoop system and the prefetching-enabled Hadoop system (PSP) processing a single large file.



(b) Performance improvement offered by the prefetching-enabled Hadoop system (PSP) processing a single large file.



(c) The response time comparison between the native Hadoop system and the prefetching-enabled Hadoop system (PSP) processing multiple small files.



(d) Performance improvement offered by the prefetching-enabled Hadoop system (PSP) processing multiple small files.

Figure 4.7: The performance improvement of our prefetching-enabled Hadoop system (PSP) over the native Hadoop system.

results plotted in Figure 4.4 also confirm that compared with the native Hadoop, our PSP mechanism can reduce the response time of WordCount by 8.9% (for the case of 1 GB memory) and 8.1% (for the case of 1 GB memory), respectively.

4.3.3 Large vs. Small Files

To evaluate the impact of file size on the performance of Hadoop clusters, we compare the execution times of Grep and WordCount processing both a single large file and multiple small

files. Keeping the input data amount fixed, we test two different types of data configuration for both Grep and WordCount. In this first configuration, the input data is a single 1 GB file. In the second case, we divide this 1GB input file into 1000 small files of equal size (i.e., the size of each small file is 1MB).

Figure 4.5 shows that although the total data amount (i.e., 1GB) for both configurations is the same, our predictive scheduling and prefetching scheme (PSP) offers different performance improvements for the two data configurations. Specifically, our PSP approach reduces the response time of Grep by 9.1% for the first case of a single large file and 18% for the second case of multiple small files. PSP also shortens the response time of WordCount by 8.3% for the single large file and 24% for the multiple small files. Hadoop applications processing a large number of small files benefit extremely well from our PSP approach, because accessing small files in HDFS is very slow.

The experimental results plotted in Figure 4.5 strongly suggest that regardless of the tested Hadoop benchmarks, our PSP scheme can significantly improve the performance of clusters for Hadoop applications processing a huge collection of small files. In the worst case scenario where there is a single large input file, the PSP scheme is able to achieve at least 8.1% performance improvement in terms of reducing execution times of Hadoop applications.

4.3.4 Hadoop Clusters

We run multiple applications on the Hadoop cluster to quantify the performance of our predictive scheduling and prefetching scheme (PSP) in a real-world setting. Table 4.2 summarizes the characteristics of the tested cluster.

Table 4.2: The Test Sets in Experiments

number	1	2	3	4
Workload	WordCount	WordCount	WordCount	Grep
nodes	3	6	12	12
Input file size	15GB	15GB	15GB	15GB
Spilt data size	64MB	64MB	64MB	64MB

Figure 4.7(a) shows the overall performance improvement achieved by PSP-enabled Hadoop cluster processing a single large file. We observe from Figures 4.7(a) and 4.7(b) that PSP noticeably improves the performance of the tested Hadoop cluster. For example, PSP reduces the average execution time for both Hadoop benchmarks on the cluster by an average of 9% for the single-large-file case. This performance improvement is very similar to the previous single-machine case.

Figure 4.7(c) illustrates the average execution times of the benchmarks processing a large collection of small files on the PSP-enabled cluster and the native Hadoop cluster. Figures 4.7(c) and 4.7(d) demonstrate the performance improvements offered by our PSP scheme. The results confirm that PSP can significantly improve a Hadoop cluster's performance when the cluster is running Hadoop applications processing a huge set of small files. For example, PSP is capable of shortening the average execution times of the two Hadoop benchmarks by an average of 25%.

4.4 Summary

In this part of the dissertation study, we observed that the task processing procedure in Hadoop may introduce data transfer overhead in a cluster. Realizing that the data transfer overhead is caused by the data locality problem in Hadoop, we proposed a predictive scheduling and prefetching mechanism or PSP for short to hide data transfer overhead. Our PSP mechanism seamlessly integrate a prefetching module and a prediction module with the Hadoop's job scheduler. The prediction module proactively predicts subsequent blocks to be accessed by computing nodes in a cluster; whereas the prefetching module preloads these future blocks in the cache of the nodes. The proposed PSP is able to avoid I/O stalls incurred by predicting and prefetching data blocks to be accessed in the future.

We tested the execution times of two Hadoop benchmarks runing on a 10-node cluster, where the proposed PSP mechanism is incorporated into the Hadoop system. The empirical results show that PSP noticeably improves the performance of the Hadoop cluster by an

average of 9% for the single-large-file case and by an average of 25% for the multiple-small-files case. This study shows that Hadoop applications processing a huge collection of small files benefit extremely well from our PSP approach, because accessing small files in HDFS is very slow.

We demonstrated that the performance of the tested Hadoop cluster is very sensitive to main memory size. In particular, our results suggest that a larger input file makes the Hadoop cluster more sensitive to the memory size. This study confirms that in addition to applying the PSP scheme, increasing memory size is an efficient way to improve the performance of Hadoop cluster processing large files.

Chapter 5

Preshuffling

In the previous Chapter we present a predictive scheduling and prefetching mechanism to improve the performance of Hadoop Clusters by hiding data transfer overhead. In this Chapter, we focus on a new reshuffling scheme to further improve Hadoop's system performance.

5.1 Motivations for a New Preshuffling Scheme

5.1.1 Shuffle-Intensive Hadoop Applications

Recall that a Hadoop application has two important phases - map and reduce. The execution model of Hadoop can be divided into two separate steps. In the first step, a map task loads input data and generates some ¡key,value¿ pairs. In this step, multiple map tasks can be executed in parallel on multiple nodes in a cluster. In step two, all the pairs for a particular key are pulled to a single reduce task after the reduce task communicates and checks all the map tasks in the cluster.

Reduce tasks depend on map tasks; map tasks are followed by reduce tasks. This particular sequence prevents reduce tasks from sharing the computing resources of a cluster with map tasks, because there is no parallelism between a pair of map and reduce tasks. During an individually communication between a set of map tasks and a reduce task, an amount of intermediate data (i.e., result generated by the map tasks) is transferred from the map tasks to the reduce task through the network interconnect of a cluster. This communication between the map and reduce tasks is also known as the shuffle phase of a Hadoop application.

In an early stage of this study, we observe that a Hadoop application's execution time is greatly affected by the amount of data transferred during the shuffle phase. Hadoop

applications generally fall into two camps, namely, non-shuffle-intensive and shuffle-intensive applications. Non-shuffle-intensive applications transfer a small amount of data during the shuffle phase. For instance, compared with I/O-intensive applications, computation-intensive applications may generate a less amount of data in shuffle phases. On the other hand, shuffle-intensive applications move a large amount of data in shuffle phases, imposing high network and disk I/O loads. Typical shuffle-intensive applications include the inverted-index tool used in search engines and the k-means tool applied in the machine learning field. These two applications transfer more than 30% data through network during shuffle phases.

5.1.2 Alleviate Network Load in the Shuffle Phase

In this Chapter, we propose a new shuffling strategy in Hadoop to reduce heavy network loads caused by shuffle-intensive applications. The new shuffling strategy is important, because network interconnects in a Hadoop cluster is likely to become a performance bottleneck when the cluster is shared among a large number of applications running on virtual machines. In particular, the network interconnects become scarce resource when many shuffle-intensive applications are running on a Hadoop cluster in parallel.

We propose the following three potential ways of reducing network loads incurred by shuffle-intensive applications on Hadoop clusters.

1. First, decreasing the amount of data transferred during the shuffle phase can effectively reduce the network burden caused by the shuffle-intensive applications. To reduce the amount of transferred data in the shuffle phase, combiner functions can be applied to local outputs by map tasks prior to storing and transferring intermediate data. This strategy can minimize the amount of data that needs to be transferred to the reducers and speeds up the execution time of the job.

2. Second, there is no need for reduce tasks to wait for map tasks to generate an entire intermediate data set before the data can be transferred to the reduce tasks. Rather, a

small portion of the intermediate data set can be immediately delivered to the reduce tasks as soon as the portion becomes available.

3. Third, heavy network loads can be hidden by overlapping data communications with the computations of map tasks. To improve the throughput of the communication channel among nodes, intermediate results are transferred from map tasks to reduce tasks in a pipelining manner. Our preliminary findings show that shuffle time is always much longer than map tasks' computation time; this phenomenon is especially true when network interconnects in a Hadoop cluster are saturated. A pipeline in the shuffle phase can help in improving throughput of Hadoop clusters.

4. Finally, map and reduce tasks allocated within a single computing node can be coordinated in a way to have their executions overlapped. Overlapping these operations inside a node can efficiently shorten the execution times of shuffle-intensive applications. A reduce task checks all available data from map nodes in a Hadoop cluster. If reduce and map tasks can be grouped with particular key-value pairs, network loads incurred in the shuffle phase can be alleviated.

5.1.3 Benefits and Challenges of the Preshuffling Scheme

There are three benefits of our preshuffling scheme:

- Data movement activities during shuffle phases is minimized.
- Long data transfer times are hidden by a pipelining mechanism.
- Grouping map and reduce pairs to reduce network load.

Before obtaining the above benefits from the preshuffling scheme, we face a few design challenges. First, we have to design a mechanism allowing a small portion of intermediate data to be periodically transferred from map to reduce tasks without waiting an entire intermediate data set to be ready. Second, we must design a grouping policy that arranges

map and reduce tasks within a node to shorten the shuffle time period by overlapping the computations of the map and reduce tasks.

5.1.4 Chapter Organization

The rest of the Chapter is organized as follows. Section 5.2 describes the design of our preshuffling algorithm after presenting the system architecture. Section 5.3 presents the implementation details of the preshuffling mechanism in the Hadoop system. In Section 5.4, we evaluate the performance of our preshuffling scheme. Finally, Section 5.5 concludes this chapter.

5.2 Design Issues

In this section, we first present the design goals of our preshuffling algorithm. Then, we describe how to incorporate the preshuffling scheme into the Hadoop system. We also show a way of reducing the shuffling times of a Hadoop application by overlapping map and reduce operations inside a node.

5.2.1 Push Model of the Shuffle Phase

A typical reduce task consists of three phases, namely, the shuffle phase, the sort phase, and the reduce phase. After map tasks generate intermediate (key, value) pairs, reduce tasks fetch in the shuffle phase the (key, value) pairs. In the shuffle phase, each reduce task handles a portion of the key range divided among all the reduce tasks. In the sort phase, records sharing the same key are groups together; in the reduce phase, a user-defined reduce function is executed to process each assigned key and its list of values.

To fetch intermediate data from map tasks in the shuffle phase, HTTP requests are issued by a reduce task to five (this default value can be configured) number of TaskTrackers. The locations of these TaskTrackers are managed by the JobTracker located in the Master node of a Hadoop cluster. When a map or reduce TaskTracker finishes, the TaskTracker

sends a heartbeat to the JobTracker in the master node, which assigns a new task to the TaskTracker. The master node is in charge of determining time when reduce tasks start running and data to be processed. Map task and reduce tasks are stored in two different queues.

Reduce tasks pull intermediate data (i.e., (key, value) pairs) from each TaskTracker that is storing the intermediate data. In this design, application developers can simply implement separate map tasks and reduce tasks without dealing with the coordination between the map and reduce tasks. In the shuffle phase the above pull model is not efficient, because reduce tasks are unable to start their execution until the intermediate data are retrieved. To improve the performance of the shuffle phase, we change the pull model into a push model. In the push model, map tasks automatically push intermediate data in the shuffle phase to reduce tasks. Map tasks start pushing (key, value) pairs to reduce tasks as soon as the pairs are produced.

We refer to the above new push model in the shuffle phase as the preshuffling technique. In what follows, we describe the design issues of our preshuffling scheme that applies the push model in the shuffle phase.

5.2.2 A Pipeline in Preshuffling

When a new job submitted to a Hadoop cluster, the JobTracker assigns map and reduce tasks to available TaskTrackers in the cluster. Unlike the pulling model, the pushing model of preshuffling push intermediate data produced by map tasks to reduce tasks. The preshuffling scheme allows the map tasks to determine a partition records to be transferred a reduce task. Upon the arrival of the partition records, the reduce task sorts and stores these records into the node hosting the reduce task. Once the reduce task is informed that all the map tasks have been completed, the reduce task performs a user-defined function to process each assigned key and its list of values. The map tasks continue generating intermediate records to be delivered the reduce tasks.

Let us consider a simple case where a cluster has enough free slots allowing all the tasks of a job to run after the job is submitted to the cluster. In this case, we establish communication channels between a reduce task and all the map tasks pushing intermediate data to the reduce task. Since each map task decides reduce tasks to which the intermediate data should be pushed, the map task transfers the intermediate data to the corresponding reduce tasks immediately after the data are produced by the map task.

In some cases, there might not be enough free slots available to schedule every task in a new Hadoop job. If a reduce task can not be executed due to limited number of free slots, map tasks can store intermediate results in memory buffers or local disks. After a free slot is assigned to the reduce task, the intermediate results buffered in the map tasks can be sent to the reduce task.

Shuffle phase time in many cases is much longer than map phase time (i.e., tasks' computation time); this problem is more pronounced true when network interconnects are scarce resource in a Hadoop cluster. To improve the performance of the preshuffling scheme, we build a pipeline in the shuffle phase to proactively transfer intermediate data from map tasks to reduce tasks. The pipeline aims at increasing the throughput of preshuffling by overlapping data communications with the computations of map tasks.

We design a mechanism to create two separate threads in a map task. The first thread processes input data, generates intermediate records, and completes the sort phase. The second thread manages the aforementioned pipeline that sends intermediate data from map tasks to reduce tasks immediately when the intermediate outputs are produced. The two threads can work in parallel in a pipelining manner. In other words, the first thread implements the first stage of the pipeline; the second thread performs the second stage of the pipeline. In this pipeline, the first stage is focusing on producing intermediate results to be stored in the memory buffers, whereas the second stage periodically retrieves the intermediate results from the buffers and transfers the results to the connected reduce tasks.

5.2.3 In-memory Buffer

The push model does not require reduce tasks to wait a long time period before map tasks complete the entire map phase. Nevertheless, pushing intermediate data from map to reduce tasks in the preshuffling phase is still a time-consuming process. The combiner process in a map task is an aggregate function (a reduce-like function) that groups multiple distinct values together as input to form a single value. If we plan to implement the preshuffling mechanism to directly send intermediate outputs from map to reduce tasks, we will have to ignore the combiner process in map tasks. In the native Hadoop system, the combiner can help map tasks to illuminate relevant data, thereby reducing data transfer costs. Sending all the data generated from map tasks to reduce tasks increases response time and downgrades the performance of Hadoop applications. Without the pre-sorting and filtering process in the combiner stage, reduce tasks should spend much time in sorting for merging values.

Instead of sending an entire buffered content to reduce tasks directly, we design a buffer mechanism to temporary collect intermediate data. The buffer mechanism immediately sends a small portion of the intermediate data to reduce tasks as soon as the portion is produced. A configurable threshold is used to control the size of the portion. Thus, once the size of buffered intermediate results reaches the threshold, the map task sorts the intermediate data based on reduce keys. Next, the map task writes the buffer to its local disk. Then, the second stage of the pipeline is invoked to check whether reduce tasks have enough free slots. If nodes hosting reduce tasks are ready, a communication channel between the map and reduce tasks are established. The combined data produced in the first stage of the pipeline can be passed to reduce tasks in the second stage of the pipeline.

In cases where nodes hosting reduce tasks are not ready, the second stage of the pipeline will have to wait until the reduce tasks are available to receive the pushed data. This pipeline mechanism aims to improve the throughput of the shuffling stage, because the pipeline makes it possible for map tasks to send intermediate data as soon as a portion of the data is produced by map functions.

In the design of our preshuffling scheme, it is flexible to dynamically control the amount of data pushed from map to reduce tasks by adjusting the buffer's threshold. A high threshold value means that each portion to be pushed from map tasks in the second stage of the pipeline is large; a small threshold value indicates that each portion shipped to reduce tasks is large. If network interconnects are not overly loaded, map tasks may become a performance bottleneck. This bottleneck problem can be addressed by increasing the buffer's threshold so that each data portion pushed to reduce tasks is large. A large threshold is recommended for Hadoop clusters with fast network interconnects; a small threshold is practical for Hadoop clusters where networks are a performance bottleneck.

5.3 Implementation Issues in Preshuffling

In Hadoop, reduce tasks will not start their executions until entire intermediate output of all map tasks have been produced, although some map tasks may generate some intermediate results earlier than the other map tasks. In our preshuffling scheme, map tasks do not need to be synchronized in the way to produce a group of intermediate data to be sent to reduce tasks at the same time. Thus, a reduce task can immediately receive corresponding intermediate data generated by map tasks. However, the reduce task is unable to apply the reduce function on the intermediate data until all the date produced by every map task become available. Like reduce tasks, a Hadoop job must wait for all map tasks to finish before producing a final result.

As described in Section 5.2, a map task consists of two phases: map and map-transfer. The map phase processes an entire input file, sorts intermediate results, and then sends them to an output buffer. The sort phase in the map task groups records sharing the same key together; this group procedure otherwise should be performed in the reduce phase. In the map-transfer phase, intermediate data is transferred from buffer in map tasks to reduce tasks.

A reduce task consists two main phases - shuffle and reduce. In the shuffle phase, the reduce task not only receives its portion of intermediate output from each map task, but also performs a merge sort on the intermediate output from map tasks. In reduce tasks, the shuffle phase time accounts for a majority of the total reduce tasks' execution time. For example, 70% of a reduce task's time is spent in the shuffle phase. The shuffle phase is time consuming, because a large amount of intermediate output from map tasks must be merged and sorted in this phase. To improve the performance of the shuffle phase, we implement a preshuffling scheme where intermediate data are immediately merged and sorted when the data are produced by map tasks. After receiving required intermediate data from all map tasks, the reduce task performs a final merge sort function based on intermediate output produced by the preshuffling scheme. When the reduce task completes its final merge sort, the task reaches the reduce phase.

In a Hadoop cluster, a master node monitors the progress of each task's execution. When a map task starts its execution, the master node assigns a progress score anywhere in the range between 0 and 1. The value of a progress score is assigned based on how much of the input data the map task has processed [**?**]. Similarly, we introduce a progress score, allowing the preshuffling scheme to monitor the progress of reduce tasks. Progress scores of reduce tasks are assigned based on how much intermediate data of each portion has been consumed by the reduce tasks. The progress score is incorporated with the data structure of intermediate data. Thus, when a partition of intermediate file is transferred to a reduce task, the progress score of this partition is also received by the reduce task. The average progress score of all relevant partitions in each intermediate data file can be considered as the progress of a reduce task.

Each node hosting reduce tasks individually runs the tasks. In heterogeneous Hadoop clusters, nodes may run tasks at different speed. Once a reduce task has made sufficient progress, the task reports its progress score written to a temporary file on HDFS. For example, we can set several granularity; the user can set the default value as 20%, 40%, 60%, 80%,

and 100%. When reduce progress reaches this value, the progress score will be automatically written down to HDFS.

By aggressively pushing data from map tasks to reduce tasks, the push model can increase the throughput of the Hadoop system by partially overlapping communication and transfer times among the map and reduce tasks. The preshuffling scheme, when used in combination with the push model, can boost the performance of Hadoop clusters. The performance improvement offered by preshuffling and the push model becomes more pronounced when network interconnection is a performance bottleneck of the clusters.

5.4 Evaluation performance

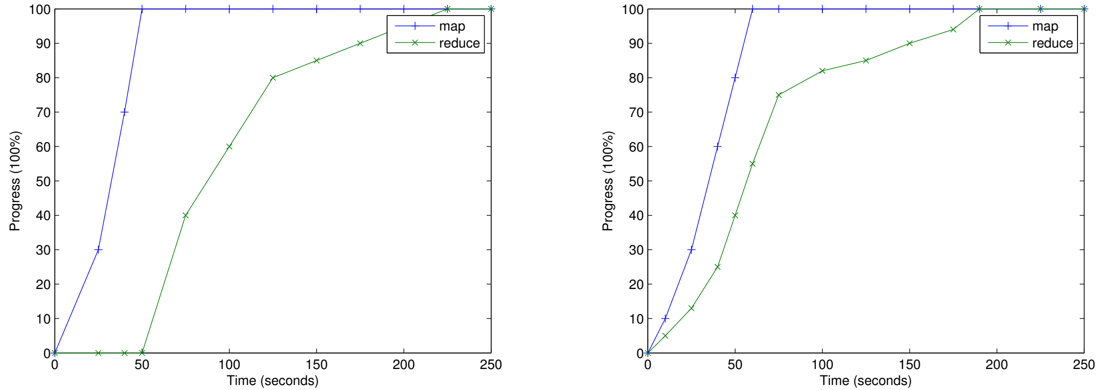
5.4.1 Experimental Environment

To evaluate the performance of the proposed preshuffling scheme incorporated in the push model with a pipelining technique, we run Hadoop benchmarks on a 10-node cluster. Table 5.1 summarizes the configuration of the cluster used as a testbed for the performance evaluation. Each computing node in the cluster is equipped with two dual-core 2.4 GHz Intel processors, 2GB main memory, 146 SATA hard disk, and a Gigabit Ethernet network interface card.

Table 5.1: Test Bed

CPU	Intel Xeon 2.4GHz
Memory	2GB Memory
Disk	SEGATE 146GB
Operation System	Ubuntu 10.4
Hadoop version	0.20.2

In our experiments, we vary the block size in HDFS to evaluate the impacts of block size system performance. In this study, we focus on impact of preshuffling and the push model on Hadoop and; therefore, we disable the data replica feature of HDFS. Nevertheless, using the preshuffling mechanism in combination with the data replica mechanism can significantly improve performance of Hadoop clusters.



(a) The execution time of WordCount processing 1GB data on the native Hadoop system is 450 seconds.

(b) The execution time of WordCount processing 1GB on the preshuffling-enabled Hadoop system is 380 seconds.

Figure 5.1: The progress trend of WordCount processing 1GB data on the 10-node Hadoop cluster.

In this part of the dissertation study, we test the following two Hadoop benchmarks running on the cluster, in which the preshuffling scheme is integrated with the push model to improve the performance of the shuffle phase in Hadoop applications.

1. WordCount (WC): This Hadoop application counts the frequency of occurrence for each word in a text file. Map tasks process different sections of input files and return intermediate data that consists of several pairs word and frequency. Then, reduce tasks add up the values for each identity word. The Word-Count is a memory-intensive application.

2. Sort: This Hadoop application puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. The output list of this application is in a non-decreasing order.

5.4.2 In Cluster

We compare the overall performance between the native Hadoop and the preshuffling-enabled Hadoop on a 10-node cluster. We measure the execution times of the two tested Hadoop benchmarks running on the Hadoop cluster, where the default block size is 64 MB.

Figure 5.4.2 illustrates the progress trend of WordCount processing 1GB data on the native Hadoop. The progress trend shown in Figure 5.4.2 indicates how the map and reduce tasks are coordinating. For example, Figure 5.1(a) shows that in the native Hadoop system, the reduce task does not start its execution until the all the map tasks complete their executions at time 50. Figure 5.1(b) proves that in the preshuffling-enabled Hadoop, our push model makes it possible for the reduce task in WordCount to begins its execution almost immediately after the map task gets started.

Our solution shortens the execution time of WordCount by approximately 15.6%, because the reduce task under the push model receives intermediate output produced by the map tasks as soon as the output become available.

Figures 5.1(a) and 5.1(b) show that it takes 50 seconds to finish the map task in the native Hadoop and its takes about 60 seconds to complete the map in the preshuffling-enabled Hadoop. The preshuffling-enabled Hadoop system has a longer map task than the native Hadoop, because in our push model part of the shuffle phase is handled by the map task rather than the reduce task in the native Hadoop. Forcing the map task to process the preshuffling phase is an efficient way of reducing heavy load imposed on reduce tasks. As a result, the preshuffling-enabled Hadoop cluster can complete the execution of WordCount faster than the native Hadoop cluster.

5.4.3 Large Blocks vs. Small Blocks

Now we evaluate the impact of block size on the performance of preshuffling-enabled Hadoop clusters. The goal of this set of experiments is to quantify the sensitivity of our preshuffling scheme on the block size using the two Hadoop benchmarks. We run the WordCount and Sort benchmarks on both the native Hadoop and the preshuffling-enabled Hadoop clusters when the block size is set to 16MB, 32MB, 64MB, 128MB, and 256MB, respectively.

Figures 5.2 and 5.3 shows the performance improvement of the preshuffling-enabled Hadoop cluster over the native Hadoop cluster as a function of the block size. Figure 5.2

Figure 5.2: Impact of block size on the preshuffling-enabled cluster running WordCount.

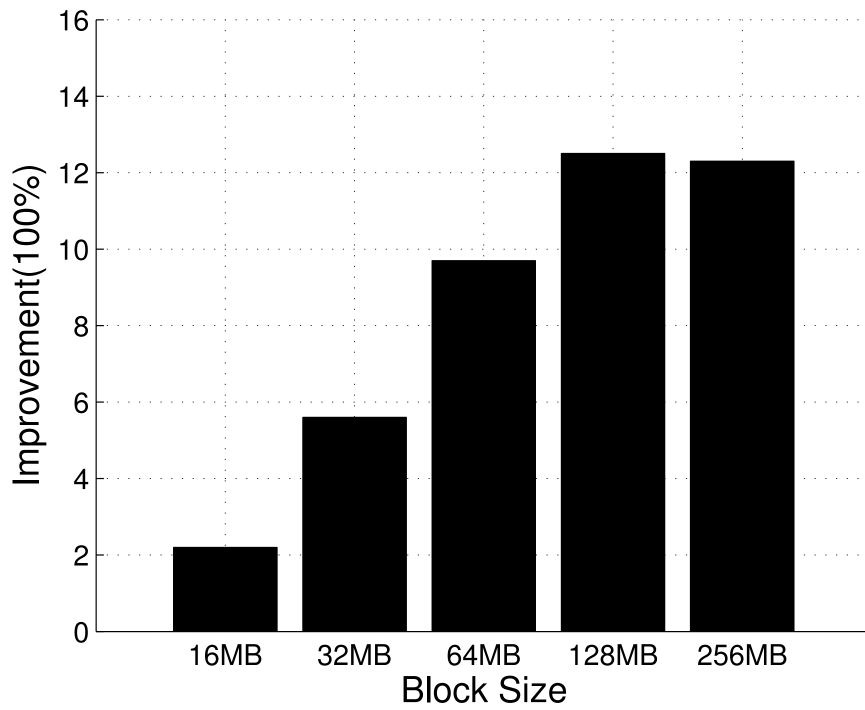
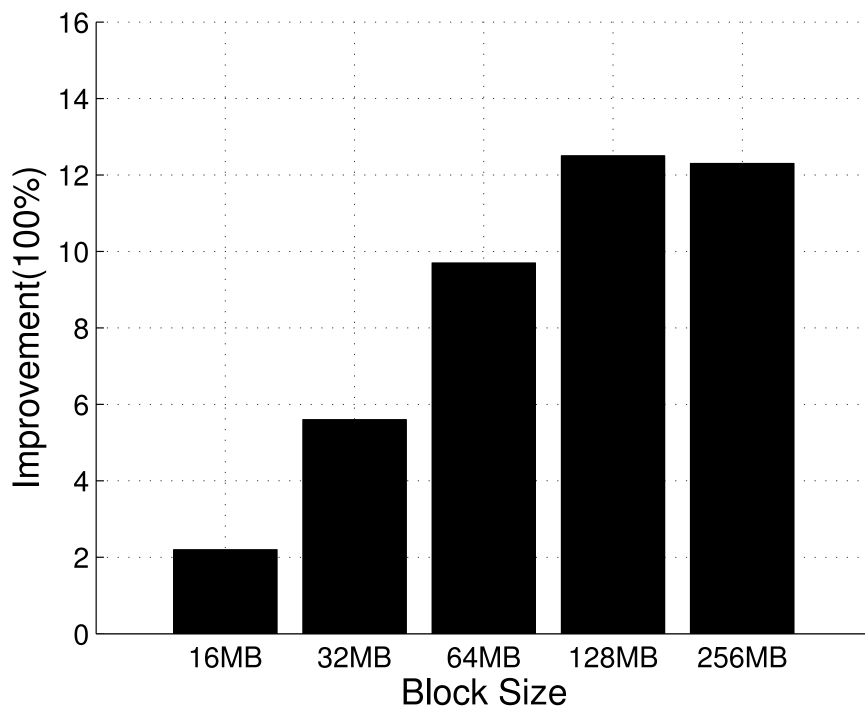


Figure 5.3: Impact of block size on the preshuffling-enabled Hadoop cluster running Sort.



demonstrates that the improvement offered by preshuffling in case the of WordCount increases when the block size goes up from 16 MB to 128 MB. However, increasing the block size from 128 MB to 256 MB does not provide a higher improvement percentage. Rather, the improvement slightly drops from 12.5% to 12.2% when the block size is changed from 128 MB to 256 MB. The experimental results plotted in Figure 5.2 suggest that a large block size allows the preshuffling scheme to offer good performance improvement. The improvement in terms of percentage is saturated when the block size is larger than 128 MB.

Figure 5.3 shows the performance improvement of preshuffling on the 10-node cluster running the Sort application. The results plotted in Figure 5.3 are consistent with those shown in Figure 5.2. For the two Hadoop benchmarks, the performance improvement offered by preshuffling is sensitive to block size when the block size is smaller than 128 MB.

5.5 Summary

A Hadoop application's execution time is greatly affected by the shuffling phase, where an amount of data is transferred from map tasks to reduce tasks. Moreover, improving performance of the shuffling phase is very critical for shuffle-intensive applications, where a large amount of intermediate data is delivered in shuffle phases. Making a high-efficient shuffling scheme is an important issue, because shuffle-intensive applications impose heavy network and disk I/O loads during the shuffle phase. In this chapter, we proposed a new push model, a new preshuffling module, and a pipelining mechanism to efficiently boost the performance of Hadoop clusters running shuffle-intensive applications.

In the push model, map tasks automatically send intermediate data (i.e., (key, value) pairs) in the shuffle phase to reduce tasks. Unlike map tasks in the traditional pull model, map tasks in the push model proactively start sending intermediate data to reduce tasks as soon as the data are produced. The push model allows reduce tasks to start their executions earlier rather than waiting until an entire intermediate data set becomes available. The push

model improves the efficiency of the shuffle phase, because reduce tasks do not need to be strictly synchronized with their map tasks waiting for the entire intermediate data set.

Our preshuffling scheme aims to release the load of reduce tasks by moving the presorting and filtering process from reduce tasks to map tasks. As a result, reduce tasks with the support of the preshuffling scheme spend less time in sorting to merge values.

In the light of the push model and the preshuffling scheme, we built a 2-stage pipeline to efficiently move intermediate data from map tasks to reduce tasks. In stage one, local buffers in a node hosting map tasks temporarily store combined intermediate data. In stage 2, a small portion of the intermediate data stored in the buffers is sent to reduce tasks as soon as the portion is produced. In the second stage of the pipeline, the availability of free slots in nodes hosting reduce tasks are checked. If there are free slots, a communication channel between the map and reduce tasks are established. In the 2-stage pipeline, the combined data produced in the first stage of the pipeline can be passed to reduce tasks in the second stage of the pipeline.

We implemented the push model along with the preshuffling scheme in the Hadoop system, where the 2-stage pipeline was incorporated with the preshuffling scheme. Our experimental results based on two Hadoop benchmarks shows that preshuffling-enabled Hadoop clusters are significantly faster than native Hadoop clusters with the same hardware configurations. For example, the push model and the preshuffling scheme powered by the 2-stage pipeline can shorten the execution times of the two Hadoop applications (i.e., WordCount and Sort) by an average of 10% and 14%, respectively.

Chapter 6

Related Work

This chapter briefly presents previous studies relevant to this dissertation research. XXXX our own from three perspectives: data locality, prefetching and pre-shuffling mechanisms.

6.1 Implementations of MapReduce

MapReduce is a distributed framework proposed by Google in 2004 for data-intensive computing on large-scale clusters [14]. MapReduce is useful in a wide range of applications including: distributed grep, distributed sort, web link-graph reversal, term-vector per host, web access log stats, inverted index construction, document clustering, complex computation [65], machine learning [8], dynamic peer-to-peer environments [41] and statistical machine translation. The MapReude framework is inspired by map and reduce functions commonly used in functional programming [32][39].

The MapReduce programming model has been adopted in various computing environments supporting a wide range of applications [53]. These computing environments include scalable computing services (e.g., the Windows AzureMapReduce system) [29], iterative computing (e.g., the Twister system) [17], memory-intensive/CPU-intensive computing environments (e.g., the LEMO-MR system) [19], multi-core/many-core systems, desktop grids [69], volunteer computing environments [40], cloud computing environments [42], and mobile computing environments [16].

MapReduce libraries have been written in C++, Erlang, Java, OCaml, Perl, Python, and other programming languages. Hadoop - implemented in the Java language - is an open source implementation of MapReduce. Hadoop becomes popular as a high-performance

computing platform for numerous data-intensive applications. A variety of techniques have been proposed to improve performances of Hadoop clusters.

Some studies has been focused on the implementation and performance evaluation of the MapReduce model [5][43]. For example, Phoenix[58][80] - a MapReduce implementation on multiprocessors - uses threads to spawn parallel Map or Reduce tasks. Phoenix also provides shared-memory buffers for map and reduce tasks to communicate without excessive data transfer overhead. The runtime system of Phoenix schedules tasks dynamically across available processors in order to balance load balance while maximizing computing throughput. Furthermore, the Phoenix runtime system automatically recovers permanent faults during task execution by repeating or re-assigning tasks.

Mars [18][20] - a MapReduce framework on graphics processors (GPUs) - aims at hiding the programming complexity of GPUs behind a simple yet powerful MapReduce interface. The Mars runtime system automatically manages task partitioning, data distribution, and parallelization on the GPUs.

6.2 Data Placement in Heterogeneous Computing Environments

Parallel File Systems. There are two types of systems handling large files in clusters, namely parallel file systems and Internet service file systems [77]. Representative parallel file systems in clusters are Lustre [1] and PVFS (Parallel Virtual File System) [56]. Hadoop distribution file system(HDFS) [13] is a popular Internet service file system that provides an abstraction for data processing in the MapReduce frameworks.

Hadoop works in combination with any distributed file system [3] that can be mounted by the underlying operating system simply by using a file:// URL; however, this feature comes at a price - the loss of locality. To reduce network traffic, Hadoop needs to manage information regarding data and servers in a cluster. For example, Hadoop must be aware of the location of data to be processed by Hadoop applications. A Hadoop-specific file system allows Hadoop to keep track of meta data information used to manage files stored on a

cluster. Hadoop Distributed File System (HDFS), Hadoop's own rack-aware file system, is designed to scale to tens of petabytes of storage and runs on top of the file systems of Linux.

Amazon S3 file system [50] is developed for clusters hosting the Amazon Elastic Compute Cloud server-on-demand infrastructure; there is no rack-awareness in Amazon's file system. CloudStore (previously known as Kosmos Distributed File System) is a rack-aware file system. CloudStore is Kosmix's C++ implementation of the Google File System. CloudStore supports incremental scalability, replication, checksumming for data integrity, client side fail-over and access from C++, Java, and Python. There exits a FUSE module that enables file systems to be mounted on Linux. The FTP File system stores all its data on remotely accessible FTP servers.

MapReduce for Heterogeneous Computing. Growing evidence shows that heterogeneity issues become important in the context of MapReduce frameworks [46]. Zaharia *et al.* implemented a new scheduler - LATE - in Hadoop to improve MapReduce performance by speculatively running tasks that significantly hurt response time [47]. Asymmetric multi-core processors (AMPs) address the I/O bottleneck issue, using double-buffering and asynchronous I/O to support MapReduce functions on clusters with asymmetric components [46]. After classifying MapReduce workloads into three categories based on CPU and I/O utilization [70], Chao *et al.* designed the Triple-Queue Scheduler in light of the dynamic MapReduce workload prediction mechanism or MR-Predict.

The major difference between our data placement solutions (see Chapter 3) and the aforementioned techniques for heterogeneous MapReduce frameworks is that our schemes take into account data locality and aim to reduce data transfer overheads.

6.3 Prefetching

Prefetching mechinism An array of prefetching techniques have been proposed to improve the performance of main memory in computing systems [68][72][11]. Cache prefetching

techniques used to improve effectiveness of cache-memory systems have been widely explored for a variety of hardware and software platforms [85][54].

An increasing number of computing systems are built to support multimedia applications; various prefetching mechanisms (see, for example, [12][82]) were developed to improve performance of multimedia systems. A few studies were focused on prefetching approaches to boosting I/O performance in computer systems [6][37][49].

Many existing prefetching solutions were designed specifically for local file systems. In contrast, the prefetching scheme (see Chapter 4) proposed in this dissertation is tailored for a distributed file system supporting Hadoop clusters.

Scheduling Algorithms Performance of Hadoop systems can be improved by efficiently scheduling tasks on Hadoop clusters. Many scheduling algorithms might be adopted in Hadoop. For example, the FairScheduler and Capacity Scheduler provide more opportunity for later jobs to get scheduled. Zaharia *et al.* [47] implemented a new scheduling algorithm called LATE (i.e., Longest Approximation Time to End) in Hadoop to improve Hadoop system performance by speculatively executing tasks that decrease response time the most. The dynamic proportional scheduler [63][36] provides job sharing and prioritization capabilities in cluster computing systems, thereby allowing multiple jobs to share resources and services in clusters.

Meng *et al.* studied an approach to estimating time and optimizing performance for Hadoop jobs. Meng's scheduling solution aims at minimizing total completion time of a set of MapReduce jobs [2]. Kristi *et al.* [45] estimates the progress of queries that run as MapReduce task graphs. Most efforts in scheduling for Hadoop clusters focus on handling various priorities; most efforts in estimating time in Hadoop pay attention to runtime estimation of running jobs.

A recent study that is closely related to this dissertation research can be found in [55], where a scheduler was proposed to increase system resource utilization by attempting to satisfy time constraints associated with jobs. The scheduler presented in [55] does not consider

the schedulability of a job prior to accepting it for execution. This scheduler emphasizes map tasks and; therefore, reduce tasks are not considered in the scheduler.

The scheduler in the native Hadoop system uses a simple FIFO (First-In-First-Out) policy. Zaharia *et al.* [22] proposed the FAIR scheduler optimized for multi-user environments. The FAIR scheduler works very well on a single cluster shared among a number of users, because FAIR reduces idle times of short jobs to offer fast response times for the short jobs. However, scheduling decisions made by FAIR are not dynamically adapted based on job progress, making FAIR inadequate for applications with different performance goals [83].

In a recent study, Sandholm and Lai developed a mechanism to dynamically assign resources of a shared cluster to multiple Hadoop instances [62][21]. In their approach, priorities are defined by users using high-level policies such as a market account. The users can independently determine the priorities of their jobs; the system allocates running times according a spending rate. If the account balance of a user reaches zero, no further tasks of that user are assigned to the cluster.

Attention has been paid to the data locality issue in the MapReduce computing platforms. For example, Seo proposed the prefetching and pre-shuffling scheme or HPMR to improve the performance in a shared MapReduce computing environment [64]. HPMR contains a predictor module that helps to make optimized scheduling decisions. The predictor module was integrated with a prefetching scheme, thereby exploiting data locality by transferring data from a remote node to a local node in a pipelining manner [47].

Our preshuffling approach described in Chapter 5 is very different from the HPMR scheme in the sense that our solution relies on a pipeline built inside a node hosting map tasks to improve performance, whereas HPMR aims at boosting performance by the virtue of a data communication pipeline between a pair of two nodes.

6.4 Shuffling and Pipline

Shuffling. Duxbury *et al.* built a theoretical model to analyze the impacts of MapReduce on network interconnects [59]. There are two new findings in their study. First, during the shuffle phase, each reduce task communicates with all map tasks in a cluster to retrieve required intermediate data. Network load is increased during the shuffle phase due to intermediate data transfers. Second, at the end reduce phase, final results of the Hadoop job is written to HDFS. Their study shows evidence that the shuffle phase can cause high network loads. Our experimental results confirm that 70% of a reduce task's time is spent in the shuffle phase. In this dissertation study (see Chapter 5), we propose a preshuffling scheme combined with a push model to release the network burden imposed by the shuffle phase.

Hoefler *et al.* implement a MapReduce runtime system using MPI (Message Passing Interface) [31]. Redistribution and reduce phases are combined in their implementation, which can benefit applications with a limited number of intermediate keys produced in the map phase.

Pipeline. Dryad [33] and DryadLINQ [81] offer a data-parallel computing framework that is more general than MapReduce. This new framework enables efficient database joins and automatic optimizations within and across MapReductions using techniques similar to query execution planning. In the Dryad-based MapReduce implementation, outputs produced by multiple map tasks are combined at the node level to reduce the amount of data transferred during the shuffle phase. Compared with this combining technique, partial hiding latencies of reduce tasks is more important and effective for shuffle-intensive applications. Such a latency-hiding technique may be extended to other MapReduce implementations.

Recently, researchers extended the MapReduce programming model to support database management systems in order to process structured files [28]. For example, Olston *et. al* developed the Pig system [26], which is a high-level parallel data processing platform integrated with Hadoop. The Pig infrastructure contains a compiler that produces sequences of Hadoop programs. Pig Latin - a textual language - is the programming language used in Pig. The

Pig Latin language not only makes it easy for programmers to implement embarrassingly parallel data analysis applications, but also offer performance optimization opportunities.

Graefer extended the Volcano query processing system to support parallelisms [30]. Exchange operators encapsulate all parallelism issues and; therefore, the parallel Volcano system makes it easy and robust to implement parallel database algorithms. Compared with MapReduce, the parallel volcano system lacks a feature of flexible scheduling.

Chapter 7

Conclusions and Future Work

In this dissertation, we have developed a number of new techniques to improve performance of Hadoop clusters. This chapter concludes the dissertation by summarizing the contributions and describing future directions. The chapter is organized as follows: Section 7.1 highlights the main contributions of the dissertation. In Section7.2, we concentrate on some future directions, which are extensions of our past and current research on Hadoop clusters.

7.1 Conclusions

We identified a set of performance problems in the Hadoop systems running on clusters. Motivated by the performance issues, we investigated three techniques to boost performance of Hadoop clusters. The solutions described in this dissertation include data placement strategies for heterogeneous Hadoop clusters, predictive scheduling/prefetching for Hadoop, and a preshuffling mechanism on Hadoop clusters.

7.1.1 Data distribution mechanism

We observed that data locality is a determining factor for Hadoop's performance. To balance workload, Hadoop distributes data to multiple nodes based on disk space availability. Such data placement strategy is very practical and efficient for a homogeneous environment where nodes are identical in terms of both computing and disk capacity. In homogeneous computing environments, all the nodes have identical workload, assuming that no data needs to be moved from one node into another. In a heterogeneous cluster, however, a high-performance node tends to complete local data processing faster than a low-performance

node. After the fast node finishes processing data residing in its local disk, the node has to handle unprocessed data in a remote slow node. The overhead of transferring unprocessed data from slow nodes to fast peers is high if the amount of moved data is huge.

An approach to improving MapReduce performance in heterogeneous computing environments is to significantly reduce the amount of data moved between slow and fast nodes in a heterogeneous cluster. To balance data-processing workload in a heterogeneous Hadoop cluster, we were motivated to develop data placement schemes, which aim to partition a large data set into data fragments that are distributed across multiple heterogeneous nodes in a cluster. Thus, the new mechanism distributes fragments of an input file to heterogeneous nodes based on their computing capacities.

Our data placement mechanism in the Hadoop distributed file system (HDFS) initially distributes a large data set to multiple nodes in accordance to the computing capacity of each node. We implemented a data reorganization algorithm in addition to a data redistribution algorithm in HDFS. The data reorganization and redistribution algorithms implemented in HDFS can be used to solve the data skew problem due to dynamic data insertions and deletions.

Our approach significantly improves performance of Hadoop heterogeneous clusters. For example, the empirical results show that our data placement mechanism can boost the performance of the two Hadoop applications (i.e., Grep and WordCount) by up to 33.1% and 10.2% with averages of 17.3% and 7.1%, respectively.

7.1.2 Predictive Scheduling and Prefetching

In an earlier stage of this dissertation study, we observed that CPU and I/O resources in a Hadoop cluster are underutilized when the cluster is running on data-intensive applications. In Hadoop clusters, HDFS is tuned to support large files. For example, typically file sizes in the HDFS file system range from gigabytes to terabytes. HDFS splits large files into several small parts that are distributed to hundreds of nodes in a single cluster; HDFS stores the

index information, called meta data, to manage several partitions for each large file. These partitions are the the basic data elements in HDFS, the size of which by default is 64MB. A large block size can shorten disk seek times; however, the large block size causes data transfer times to dominate the entire processing time, making I/O stalls a significant factor in the processing time. The large block size motivates us to investigate predictive scheduling and prefetching mechanisms (see Chapter 4) that aim to boost I/O performance of Hadoop clusters.

The predictive scheduling and prefetching scheme described in Chapter 4 is an important issue, because our scheme can bridge the performance gap between ever-faster CPUs and slow disk I/Os. Simply increasing cache size does not necessarily improve the performance of CPU and disk I/Os [51]. In the MapReduce model, before a computing node launches a new application, the application relies on the master node to assign tasks. The master node informs computing nodes what the next tasks are and where the required data blocks are located. The computing nodes do not retrieve the required data and process it until assignment notifications are passed from the master node. In this way, the CPU are underutilized by waiting a long period for the notifications are available from the master node. Prefetching strategies are needed to parallelize these workloads so as to avoid idle CPU times.

High data transfer overheads are caused by the data locality problem in Hadoop. To address this problem, we presented in Chapter 4 a predictive scheduling and prefetching mechanism called PSP that aims to improve the performance of Hadoop clusters. In this part of the study, we proposed a predictive scheduling algorithm to assign tasks to DataNodes in a Hadoop cluster. Our PSP mechanism seamlessly integrate a prefetching module and a prediction module with the Hadoop's job scheduler. The prediction module proactively predicts subsequent blocks to be accessed by computing nodes in a cluster; whereas the prefetching module preloads these future blocks in the cache of the nodes.

The proposed PSP is able to avoid I/O stalls incurred by predicting and prefetching data blocks to be accessed in the future. The prefetching scheme in PSP preloads input data from local disks and place the data into the local cache of nodes as late as possible without any starting delays of new tasks assigned to the nodes.

We evaluated the performance of our PSP scheme on a 10-node cluster running two Hadoop benchmarks. The tested cluster is powered by the Hadoop system in which the proposed PSP mechanism was incorporated. The experimental results collected on the Hadoop cluster show that PSP significantly boost the performance of the Hadoop cluster by an average of 9% for the single-large-file case and by an average of 25% for the multiple-small-files case. Our study shows strong evidence that Hadoop applications processing a huge collection of small files benefit extremely well from our PSP scheme. Processing small files by Hadoop applications can take full advantage of PSP, because accessing small files in HDFS is very slow.

Interestingly, our study also shows that the performance of the Hadoop cluster is very sensitive to main memory size. The results suggest that a larger input file makes the Hadoop cluster more sensitive to the memory size. Our dissertation study confirms that apart from applying the PSP scheme to improve performance of Hadoop systems, increasing memory capacity also is another way of achieving high performance of Hadoop cluster processing large files.

7.1.3 Data Preshuffling

Our preliminary results show that some Hadoop applications are very sensitive to the amount of data transferred during the shuffle phase. Hadoop applications can be generally classified into two groups - non-shuffle-intensive and shuffle-intensive applications. Non-shuffle-intensive applications transfer a small amount of data during the shuffle phase, whereas shuffle-intensive applications move a large amount of data in shuffle phases, imposing high network and disk I/O loads. For example, some Hadoop applications (e.g., the

inverted-index tool used in search engines) transfer more than 30% data through network during shuffle phases.

We proposed in Chapter 5 a new preshuffling strategy in Hadoop to reduce high network loads imposed by shuffle-intensive applications. Designing new shuffling strategies is very appealing for Hadoop clusters where network interconnects are performance bottleneck when the clusters are shared among a large number of applications. The network interconnects are likely to become scarce resource when many shuffle-intensive applications are sharing a Hadoop cluster. We implemented the push model along with the preshuffling scheme in the Hadoop system, where the 2-stage pipeline was incorporated with the preshuffling scheme.

In the push model described in Chapter 5, map tasks automatically send intermediate data in the shuffle phase to reduce tasks. The push model allows reduce tasks to start their executions earlier rather than waiting until an entire intermediate data set becomes available. The push model improves the efficiency of the shuffle phase, because reduce tasks do not need to be strictly synchronized with their map tasks waiting for the entire intermediate data set.

Apart from the push model, we also develop a 2-stage pipeline to efficiently transfer intermediate data. In the first stage, local buffers in a node hosting map tasks temporarily store combined intermediate data. In the second stage, a small portion of the intermediate data stored in the buffers is sent to reduce tasks as soon as the portion is produced. In the 2-stage pipeline, the combined data produced in the first stage of the pipeline can be passed to reduce tasks in the second stage of the pipeline.

We implemented the push model and a pipeline along with the preshuffling scheme in the Hadoop system. Using two Hadoop benchmarks running on the 10-node cluster, we conducted experiments to show that preshuffling-enabled Hadoop clusters are faster than native Hadoop clusters. For example, the push model and the preshuffling scheme powered by the 2-stage pipeline can shorten the execution times of the WordCount and Sort Hadoop applications by an average of 10% and 14%, respectively.

7.2 Future Work

During the course of developing new techniques to improve performance of Hadoop clusters, we have identified a couple of opening issues. In this section, we describe our future research studies in which we plan to address a few open issues that have not been addressed in this dissertation.

7.2.1 Small Files

The new findings from this dissertation study show that Hadoop clusters are inefficient when it comes to processing small files. The native Hadoop system was designed for handling large data sets; the default block size set in HDFS is 64MB. The following two reasons explain why Hadoop clusters are inadequate for processing small files.

First, the HDFS architecture does not support small files. In HDFS, each file registers an index file in the master node of a cluster; the data of each file is stored in DataNodes with a default size of 64MB. A large block size not only helps to reduce the amount of metadata managed by the master node, but also decrease disk seek times. When it comes to small files, both the amount of metadata and seek times are going up. For example, we intentionally divide a 1GB file into a thousand of small files. The number index files containing 150 bytes is increased by a thousand times. During the initialization phase of HDFS, all the metadata must be loaded into main memory, thereby increasing data loading time in the master node. Furthermore, accessing small files stored on disks is time consuming due to high seek time delays.

Second, extra computing time is required to process small files. Our experimental results show that processing small files takes 10 even 100 more times than processing a single large file containing the same amount of data. In the Hadoop system, map tasks always handle one block at a time. Hence, the master node needs to create and maintain a data structure to monitor each processing procedure. Moreover, before launching a new reduce task, the reduce task has to communicate with every map tasks in the cluster to acquire intermediate

data. When there are many small files, such data transfer phase becomes very inefficient due to enormous number of communications of small data items.

Hadoop archives or HAR files were introduced to HDFS to alleviate the performance problem of reading and writing small files [73]. The HAR file framework is built on top of HDFS. The HAR framework provides a command that packs small files into a large HAR file. The advantage of HAR is that all the small files in HAR files are visible and accessible. Hadoop can directly operate HAR files as input data for any Hadoop application.

Accessing HAR files is more efficient than reading many small files. However, loading a single large file is faster than reading a HAR file, because a two-level index file has to be retrieved before accessing a HAR file. Currently, there is no efficient way in Hadoop to locate small files in HARs. Furthermore, there is a lack of flexible way to modify small files achieved in a HAR file after the HAR file is created. We plan to investigate a possibility of using a virtual index structure with variable length blocks to record metadata of each files. We intend to study a mechanism where the HAR framework can modify the metadata of achieved small files without having to manipulate the data.

Another solution to improving performance of accessing small files in HDFS is sequence-File [35], which uses file names as keys and file contents as values. It is easy to implement a simple program to put several small files into a single SequenceFile to be processed as a streaming input for MapReduce applications. SequenceFiles are dividable and; therefore, MapReduce programs can break a large SequenceFiles into blocks and independently process each block. In the future, we plan to extend the sequenceFile framework to offer a flexible way to access a list all keys in a SequenceFile.

7.2.2 Security Issues

Much recent attention was paid to security issues in cloud computing [44][60]. Security issues in Hadoop can be addressed at various levels, including but not limited to, filesystem,

networks, scheduling, load balancing, concurrency control, and databases [74][75][32]. Yahoo provides basic security modules in the latest version of Hadoop in March 2011.

In the future, we plan to address the following security issues in Hadoop. First, we will design and implement an access control mechanism in HDFS. Our access control mechanism will be implemented at both the client level and the server level. Second, we will develop a module allowing users to control which Hadoop applications can access which data sets. Third, we will build a secure communication channel among map tasks and reduce tasks. Our secure communication scheme in Hadoop allow applications to securely run on a group of clusters connected by unsecured networks. Fourth, we will design a privacy preserving scheme in HDFS to protect users' private information stored in HDFS. Last, we will investigate a model that can guide us to make best tradeoffs between performance and security in Hadoop clusters.

7.3 Summary

In summary, this dissertation describes three practical approaches to improving the performance of Hadoop clusters. In the first part of our study, we showed that ignoring the data-locality issue in heterogeneous clusters can noticeably deteriorate the performance of Hadoop. We developed a new data placement scheme to place data across nodes in a way that each node has a balanced data processing load. In the second phase of the dissertation research, we designed a predictive scheduling and prefetching mechanism to preload data from local or remote disks into main memory in Hadoop clusters. Finally, we proposed a preshuffling scheme to preprocess intermediate data between the map and reduce stages, thereby increasing the throughput of Hadoop clusters.

The experimental results based on two Hadoop benchmarks running on a 10-node cluster show that our data placement, prefetching/scheduling, and preshuffling schemes adaptively balance the tasks and amount of data to improve the performance of Hadoop clusters in general and heterogeneous clusters in particular. In the future, we will seamlessly integrate

our three proposed techniques to offer a holistic way of improving system performance of Hadoop clusters.

Bibliography

[1] A scalable, high performance file system. `http://lustre.org`.

[2] Ashraf Aboulnaga, Ziyu Wang, and Zi Ye Zhang. Packing the most onto your cloud. In *Proceeding of the first international workshop on Cloud data management*, CloudDB '09, pages 25–28, New York, NY, USA, 2009. ACM.

[3] Azza Abouzeid, Kamil B. Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.

[4] F. Ahmad, S. Lee, M. Thottethodi, and TN Vijaykumar. Mapreduce with communication overlap (marco). 2007.

[5] B.He, W.Fang, Q.Luo, N.Govindaraju, and T.Wang. *Mars: a MapReduce framework on graphics processors*. ACM, 2008.

[6] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. *SIGMETRICS Perform. Eval. Rev.*, 23(1):188–197, 1995.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[8] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, pages 281–288, 2006.

[9] C.Olston, B.Reed, U.Srivastava, R.Kumar, and A.Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[10] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!s hosted data serving platform. Technical report, In Proc. 34th VLDB, 2008.

[11] R. Cucchiara, A. Prati, and M. Piccardi. Data-type dependent cache prefetching for mpeg applications. *Performance, Computing, and Communications Conference, 2002. 21st IEEE International*, 0:115–122, 2002.

[12] Rita Cucchiara. Temporal analysis of cache prefetching strategies for multimedia applications. In *Proc. of IEEE Intl. Performance, Computing and Communications Conf. (IPCCC*, pages 311–318, 2001.

[13] D.Borthakur. *The Hadoop Distributed File System: Architecture and Design*. The Apache Software Foundation, 2007.

[14] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI '04*, pages 137–150, 2008.

[15] Jeffrey Dean and Sanjay Ghemawat. System and method for efficient large-scale data processing, 06 2005.

[16] Adam Dou, Vana Kalogeraki, Dimitrios Gunopulos, Taneli Mielikainen, and Ville H. Tuulos. Misco: a mapreduce framework for mobile systems. In *Proceedings of the 3rd International Conference on PErvasive Technologies Related to Assistive Environments*, PETRA '10, pages 32:1–32:8, 2010.

[17] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM.

[18] E.Riedel, C.Faloutsos, G.Gibson, and D.Nagl. Active disks for large-scale data processing. *Computer*, 34(6):68–74, Jun 2001.

[19] Zacharia Fadika and Madhusudhan Govindaraju. Lemo-mr: Low overhead and elastic mapreduce implementation optimized for memory and cpu-intensive applications. In *CloudCom*, pages 1–8, 2010.

[20] Wenbin Fang, Bingsheng He, Qiong Luo, and Naga K. Govindaraju. Mars: Accelerating mapreduce with graphics processors. *IEEE Trans. Parallel Distrib. Syst.*, 22:608–620, April 2011.

[21] Apache Software Foundation. Dynamic priority scheduler for hadoop. `http://issues.apache.org/jira/browse/HADOOP-4768`.

[22] Apache Software Foundation. A fair sharing job scheduler. `http://issues.apache.org/jira/browse/HADOOP-3746`.

[23] Apache Software Foundation. Hadoop. `http://hadoop.apache.org/hadoop`.

[24] Apache Software Foundation. The hbase project. `http://hadoop.apache.org/hbase`.

[25] Apache Software Foundation. The hive project. `http://hadoop.apache.org/hive`.

[26] Apache Software Foundation. The pig project. `http://hadoop.apache.org/pig`.

[27] Apache Software Foundation. The zoomkeeper project. `http://hadoop.apache.org/zookeeper`.

[28] Eric Friedman, Peter Pawlowski, and John Cieslewicz. Sql/mapreduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proc. VLDB Endow.*, 2:1402–1413, August 2009.

[29] Thilina G, Tak-Lon W, Judy Q, and Geoffrey F. Mapreduce in the clouds for science. In *CloudCom*, pages 565–572, 2010.

[30] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, SIGMOD '90, pages 102–111, New York, NY, USA, 1990. ACM.

[31] Torsten Hoefler, Andrew Lumsdaine, and Jack Dongarra. Towards efficient mapreduce using mpi. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 240–249, Berlin, Heidelberg, 2009. Springer-Verlag.

[32] Shadi Ibrahim, Hai Jin, Lu Lu, Song Wu, Bingsheng He, and Li Qi. Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud. In *CloudCom*, pages 17–24, 2010.

[33] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41:59–72, March 2007.

[34] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM.

[35] Wiley K., Connolly A, Gardner J., and Krughoff S. Astronomy in the cloud: Using mapreduce for image co-addition. , 123:366–380, March 2011.

[36] Kamal Kc and Kemafor Anyanwu. Scheduling hadoop jobs to meet deadlines. *Cloud Computing Technology and Science, IEEE International Conference on*, 0:388–392, 2010.

[37] Tom M. Kroeger and Darrell D. E. Long. Design and implementation of a predictive file prefetching algorithm. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 105–118, Berkeley, CA, USA, 2001. USENIX Association.

[38] Kevin Lai, Lars Rasmusson, Eytan Adar, Li Zhang, and Bernardo A. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent Grid Syst.*, 1:169–182, August 2005.

[39] Ralf Lammel. Google's mapreduce programming model revisited. *Sci. Comput. Program.*, 68:208–237, October 2007.

[40] Heshan Lin, Xiaosong Ma, Jeremy Archuleta, Wu-chun Feng, Mark Gardner, and Zhe Zhang. Moon: Mapreduce on opportunistic environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 95–106, New York, NY, USA, 2010. ACM.

[41] Fabrizio Marozzo, Domenico Talia, and Paolo Trunfio. Adapting mapreduce for dynamic environments using a peer-to-peer model, 2008.

[42] Fabrizio Marozzo, Domenico Talia, and Paolo Trunfio. A peer-to-peer framework for supporting mapreduce applications in dynamic cloud environments. In Nick Antonopoulos and Lee Gillam, editors, *Cloud Computing*, volume 0 of *Computer Communications and Networks*, pages 113–125. Springer London, 2010.

[43] M.Isard, M.Budiu, Y.Yu, A.Birrell, and D.Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72. ACM, 2007.

[44] Mircea Moca, Gheorghe Cosmin Silaghi, and Gilles Fedak. Distributed results checking for mapreduce in volunteer computing. *Parallel and Distributed Processing Workshops and PhD Forum, 2011 IEEE International Symposium on*, 0:1847–1854, 2011.

[45] Kristi Morton, Magdalena Balazinska, and Dan Grossman. Paratimer: a progress indicator for mapreduce dags. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 507–518, New York, NY, USA, 2010. ACM.

[46] M.Rafique, B.Rose, A.Butt, and D.Nikolopoulos. Supporting mapreduce on large-scale asymmetric multi-core clusters. *SIGOPS Oper. Syst. Rev.*, 43(2):25–34, 2009.

[47] M.Zaharia, A.Konwinski, A.Joseph, Y.zatz, and I.Stoica. Improving mapreduce performance in heterogeneous environments. *In OSDI'08: 8th USENIX Symposium on Operating Systems Design and Implementation*, October 2008.

[48] OpenPBS.org. Torque resource manager. `http://www.clusterresources.com/pages/products/torque-resource-manager.php`.

[49] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using predictive prefetching to improve world wide web latency. *SIGCOMM Comput. Commun. Rev.*, 26(3):22–36, 1996.

[50] Mayur R. Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon s3 for science grids: a viable solution? In *Proceedings of the 2008 international workshop on Data-aware distributed computing*, DADC '08, pages 55–64, New York, NY, USA, 2008. ACM.

[51] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. *SIGOPS Oper. Syst. Rev.*, 29:79–95, December 1995.

[52] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 165–178, New York, NY, USA, 2009. ACM.

[53] Alysson N. Bessani Pedro Costa, Marcelo Pasin and Miguel Correia. Byzantine fault-tolerant mapreduce: Faults are not just crashes. In *CloudCom*, Athens, Greece, 2011.

[54] Andy D. Pimentel, Louis O. Hertzberger, Pieter Struik, and Pieter van der Wolf. Hardware versus hybrid data prefetching in multimedia processors: A case study. In *in the Proc. of the IEEE Int. Performance, Computing and Communications Conference (IPCCC 2000*, pages 525–531, 2000.

[55] Jorda Polo, David Carrera, Yolanda Becerra, Malgorzata Steinder, and Ian Whalley. Performance-driven task co-scheduling for mapreduce environments. In *NOMS*, pages 373–380, 2010.

[56] pvfs2.org. Parallel virtual file system, version 2. `http://www.pvfs2.org`.

[57] Jorge-Arnulfo Q.Ruiz, Christoph Pinkel, Jorg Schad, and Jens Dittrich. Raft at work: speeding-up mapreduce applications under task and node failures. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 1225–1228, New York, NY, USA, 2011. ACM.

[58] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. *High-Performance Computer Architecture, International Symposium on*, 0:13–24, 2007.

[59] Raplesf. Analyzing network load in map/reduce. `http://blog.rapleaf.com/dev/2010/08/24/analyzing-network-load-in-mapreduce`.

[60] Indrajit Roy, Hany E. Ramadan, Srinath T. V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for mapreduce, 2009.

[61] R.Pike, S.Dorward, R.Griesemer, and S.Quinlan. *Interpreting the data: Parallel analysis with Sawzall*, volume 13. IOS Press, 2005.

[62] Thomas Sandholm and Kevin Lai. Mapreduce optimization using regulated dynamic prioritization. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, pages 299–310, New York, NY, USA, 2009. ACM.

[63] Thomas Sandholm and Kevin Lai. Dynamic proportional share scheduling in hadoop. In *JSSPP'10*, pages 110–131, 2010.

[64] Sangwon Seo, Ingook Jang, Kyungchang Woo, Inkyo Kim, Jin-Soo Kim, and Seungryoul Maeng. Hpmr: Prefetching and pre-shuffling in shared mapreduce computation environment. In *Proceedings of 11th IEEE International Conference on Cluster Computing*, pages 16–20. ACM, 2009.

[65] Sangwon Seo, Edward J. Yoon, Jae-Hong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *CloudCom*, pages 721–726, 2010.

[66] S.Ghemawat, H.Gobioff, and S.Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[67] Haiying Shen and Yingwu Zhu. A proactive low-overhead file replication scheme for structured p2p content delivery networks. *J. Parallel Distrib. Comput.*, 69(5):429–440, 2009.

[68] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14:473–530, September 1982.

[69] Bing Tang, Mircea Moca, Stephane Chevalier, Haiwu He, and Gilles Fedak. Towards mapreduce for desktop grid computing. In *Proceedings of the 2010 International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, 3PGCIC '10, pages 193–200, Washington, DC, USA, 2010. IEEE Computer Society.

[70] T.Chao, H.Zhou, Y.He, and L.Zha. *A Dynamic MapReduce Scheduler for Heterogeneous Workloads*. IEEE Computer Society, 2009.

[71] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience: Research articles. *Concurr. Comput. : Pract. Exper*, 17:323–356, February 2005.

[72] Steven P. Vanderwiel and David J. Lilja. Data prefetch mechanisms. *ACM Comput. Surv.*, 32:174–199, June 2000.

[73] Jason Venner. *Pro Hadoop*. Apress, 2009.

[74] Yongzhi Wang and Jinpeng Wei. Viaf: Verification-based integrity assurance framework for mapreduce. *Cloud Computing, IEEE International Conference on*, 0:300–307, 2011.

[75] Wei Wei, Juan Du, Ting Yu, and Xiaohui Gu. Securemr: A service integrity assurance framework for mapreduce. *Computer Security Applications Conference, Annual*, 0:73–82, 2009.

[76] Tom White. *Hadoop The Definitive Guide*. O'Reilly, 2009.

[77] W.Tantisiriroj, S.Patil, and G.Gibson. Data-intensive file systems for internet services: A rose by any other name ... *Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-08-114*, October 2008.

[78] Yahoo. Yahoo! launches worldis largest hadoop production application. `http://tinyurl.com/2hgzv7`.

[79] Christopher Yang, Christine Yen, Ceryen Tan, and Samuel R. Madden. Osprey: Implementing mapreduce-style fault tolerance in a shared-nothing distributed database. *Data Engineering, International Conference on*, 0:657–668, 2010.

[80] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 198–207, Washington, DC, USA, 2009. IEEE Computer Society.

[81] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[82] Daniel F. Z ucker, Michael J. Flynn, and Ruby B. Lee. A comparison of hardware prefetching techniques for multimedia benchmarks. Technical report, Stanford, CA, USA, 1995.

[83] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Job scheduling for multi-user mapreduce clusters. Technical Report UCB/EECS-2009-55, EECS Department, University of California, Berkeley, Apr 2009.

[84] Chen Zhang and H. De Sterck. CloudBATCH: A Batch Job Queuing System on Clouds with Hadoop and HBase. pages 368–375, November 2010.

[85] Daniel Zucker, Ruby B. Lee, and Michael J. Flynn. Hardware and software cache prefetching techniques for mpeg benchmarks. *IEEE Transactions on Circuits and Systems for Video Technology*, 10:782–796, 2000.