Assessment of Multiple MapReduce Strategies for Fast Analytics of Small Files

by

Fang Zhou

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
May 10, 2015

Keywords: Hadoop, MapReduce, Small Files

Approved by

Weikuan Yu, Chair, Associate Professor of Computer Science and Software Engineering
Sanjeev Baskiyar, Associate Professor of Computer Science and Software Engineering
Saad Biaz, Professor of Computer Science and Software Engineering
Levent Yilmaz, Professor of Computer Science and Software Engineering

Abstract

Hadoop, an open-source implementation of MapReduce, is used widely because of its ease of programming, scalability, and availability. Hadoop distributed file system (HDFS) and Hadoop MapReduce are two important components of Hadoop. Hadoop MapReduce is used to process the data stored in HDFS.

With the explosive development of cloud computing, increasing business and scientific application needs to take advantages of Hadoop. The sizes of files processed in Hadoop are not bound to very large files any more. Large amount of small files both in business and scientific areas are processed by MapReduce, such as document type files, bioinformatics files, geographic information files, and so on. In this situation, MapReduce performance of Hadoop is impacted severely. Although Hadoop and other frameworks provide some MapReduce strategies, they are not directly designed for small files. In addition, there is no theoretical analysis for evaluating MapReduce strategies for small files. In this paper, I conduct an analysis of existing different MapReduce strategies for small files and use theoretical and empirical methods to evaluate these strategies for processing small files. The experimental results demonstrate the correctness and efficiency of our analysis.

Acknowledgments

First of all, I would like to express my gratitude to my advisor, Prof. Weikuan Yu, for the continuous support of my master study and research. I learn a lot from his earnest, motivation, ambition, enthusiasm, patience, and immense of knowledge. His comprehensive and farsighted ideas impress me. His guidance and instructions raise me. I could not have imagined having a better advisor and mentor for my master study.

In addition, I want to thank the rest of my thesis committee: Prof. Baskiyar, Prof. Biaz, and Prof. Yilmaz for their encouragement, insightful comments and questions.

I also want to thank my fellow labmates in Parallel Architecture & System Laboratory (PASL) : Dr. Hui Chen, Dr. Jianhui Yue, Bin Wang, Cong Xu, Zhuo Liu, Yandong Wang, Teng Wang, Huansong Fu, Xinning Wang, Kevin Vasko, Michael Pritchard, Hai Pham, Bhavitha Ramaiahgari, Lizhen Shi, Yue Zhu, and Hao Zou, for the technical communication and discussion, for the sleepless night we were working together before the deadline, for the fun we were talking and playing together. Especially, I appreciate Bin Wang and Zhuo Liu's guidance and encourage for the research. PASL is a big family for me. I am so happy that I can be a member in it. The period of time spent in PASL is my forever wealth in my life.

I would like to thank my parents and parents-in-law. They always have no regrets in silence to pay and give me unconditional support whenever I need.

Last but not the least, I want to say thank you to my wife, Xiaoye Cheng. She helps me understand the real meaning of life, give me endless love and support. It is a great feeling to have her accompany with me.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the quick and continuous development of information technology, humanity enters the big data era. Every day, 2.5 quintillion bytes of data are created – so much that 90% of the data in the world today has been created in the last two years [12]. Some estimates for the data growth are as high as 50 times by the year 2020 [33]. Such large scale data is very difficult to process and analyze with relational database systems and desktop statistic softwares. Both industry and academy need some innovations to address this difficulty.

Google publishes three papers to introduce the related techniques used in its own cluster, MapReduce [25], BigTable [21], and the Google File System (GFS) [31]. The MapReduce paper, published in the OSDI conference, introduces a parallel processing system running on Google cluster. In this paper, authors present the related techniques about MapReduce programming model, data type, components of the system, fault tolerance, and so on. Map and reduce functions do not appear first in this paper. Actually, map and reduce functions are very popular functions in most functional programming languages. Map function performs filtering and sorting. The reduce function performs a summary operation. With map and reduce functions, users can easily implement multiple kinds of traditional operations, such as Wordcount, sort, and so on. When a user submits a job to the system, the Google MapReduce system firstly starts a master and some workers. According to the locality, the master assigns map jobs to workers. After map jobs are finished, the output of map jobs is stored on the disk. Reduce jobs get the required map outputs from the disks and create the final outputs. The system also provides very good fault tolerance and scalability. GFS supports high scalability and robustness of Google MapReduce. In GFS, there is one single Masternode and a large number of Chunkservers. Masternode is responsible for keeping

metadata in memory and responding to the file requests. Each Chunkserver sends a message including node status and block information to Masternode in a fixed interval.

Motivated by the success of Google MapReduce, both industry and the open source community started to develop multiple similar parallel processing frameworks. Hadoop [2], an open-source implementation of Google MapReduce, is the most popular system. It is designed to run parallel processing on thousands of computing nodes and provide a fault-tolerant and scalable storage service. In Hadoop there are two fundamental components, Hadoop MapReduce and Hadoop Distributed File System (HDFS). Hadoop MapReduce uses very similar techniques described in Google MapReduce. HDFS is also an open-source implementation of Google File System. Hadoop is not a simple copy of ideas from Google. When Hadoop was released, it was widely used not only in industry but also in academy. For example, in industry, Facebook runs the majority of analytics every day with the largest cluster in the world; in academy, NASA uses Hadoop to make an analysis of large scale climate data. Excluding lots of examples from industry and academy, we can also see the prosperity from its update frequency. The stable version of Hadoop is updated nearly every two months on average. Meanwhile there is a very active user mailing list and issue track system.

The size of target data in the cloud environment is always being considered as a very large file, such as a large log file. However, with the wide usage of Hadoop framework, the large file is not the only kind of file processed in the cloud. Hadoop can provide good performance of storing and processing for large files. However, with the wide usage of Hadoop framework, the large file is not the only kind of file stored in the cloud. In Education area, BlueSky is an E-learning cloud computing framework [27]. Most files stored in BuleSky are slides files, whose file size normally starts from tens of KB to single digital of MB. In Climatology, The Earth System Grid (ESG) at LLNL stores over 27 Terabytes climate change files. The average size of these files is 61 Megabytes [23]. In the area of Biology, some research applications create millions of files with an average size of less than 200 Kilobytes [37]. It

is very similar to the area of Astronomy. The files stored in Data Archive Server (DAS) at Fermi National Accelerator Laboratory (FNAL) are usually less than 1 Megabytes [38]. As shown in the examples above, more and more small files are stored in the HDFS.

Small files appear not only at the storage level but also at the process level. Increasing number of data analysis applications replies on Hadoop, such as remote sensing image analysis [19], web-scaled analysis for multimedia data mining [46], signal recognition [42], Astronomy [48], Meteorology [30], and Climate data analysis [29]. Facebook introduces the situation of data warehouse in [41], where they always need to process a lot of small files everyday.

However, the design of Hadoop MapReduce is mainly to run MapReduce jobs on large files. I will show the ineffectiveness and inefficiency of Hadoop for processing small files and present my assessment and optimization in the next chapters.

Chapters of my thesis are organized as follows. Chapter 2 introduces the background. Chapter 3 demonstrates the motivation in my research. Chapter 4 describes the assessment of various MapReduce strategies for small files. Chapter 5 demonstrates experimental results and analysis. Chapter 6 presents related work. In the end, I will conclude my thesis in Chapter 7.

Chapter 2

Background

In this chapter, we will first introduce the main components in Hadoop. There are three main components in Hadoop. One is Hadoop Common, which provides the necessary utilities that support the other Hadoop modules. The other is Hadoop Distributed File System (HDFS), which provides the high speed access to application data. The third component is Hadoop MapReduce, which is a system for parallel processing of large scale data. These three components cooperate together to make Hadoop work well. Then we want to present the next generation of Hadoop (Yarn), and plenty of frameworks in Hadoop ecosystem.

2.1 The main structure of HDFS

HDFS consists of a Namenode and a large number of Datanodes. Namenode takes responsibility of managing all metadata information and responding the file operations. Datanode is responsible for saving the real and replicated data. Namenode regularly receives the heartbeat from Datanodes in a fix time interval, 3 seconds by default. In heartbeat, each Datanode sends itself status to Namenode including the whole size, used size, and some other information. After Datanode receives the heartbeat message, Datanode updates its metadata, which is kept in memory. If Namenode cannot get any heartbeat from one Datanode in 10 minutes (default interval), Namenode will remove this Datanode from the live node list and add it into dead node list. Namenode is a "Single Point of Failure" for HDFS cluster. It means that once the Namenode goes down, the whole file system goes offline. In order to avoid this problem, HDFS permits users to start a secondary NameNode, which should run in another node in the cluster. Secondarynamenode only creates a checkpoint of namespace without adding any other redundant. To some extent, secondarynamenode

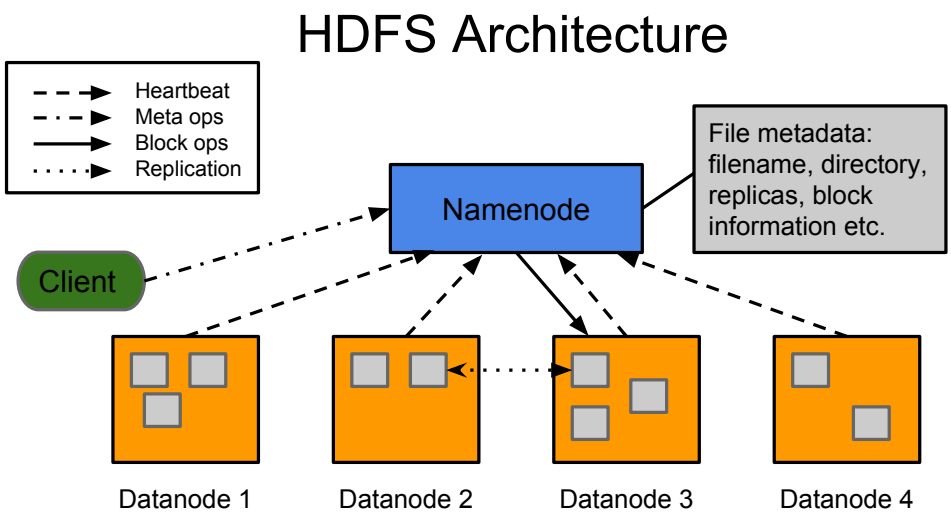


Figure 2.1: HDFS Architecture

indeed helps improve the availability of HDFS. With the development of HDFS, there are a lot of new features implemented in the latest version HDFS. The most exciting one is the range of media has been expanded. Before HDFS only can use hard disk as media. However, in the latest version, HFDS can support not only SSD but also memory. The details of HDFS architecture can be seen in Figure 2.1.

2.2 The main structure of Hadoop MapReduce

Hadoop MapReduce is the data parallel processing framework in Hadoop. It has the similar typology structure with HDFS. In Hadoop MapReduce, there are a master node named JobTracker and multiple slave nodes named TaskTracker. JobTracker is responsible for processing and scheduling the submitted application. Once a user submits a job to Hadoop, JobTracker firstly receives the submission. Then JobTracker try to communicate with Namenode to get the locality of the corresponding application data. According to a specific scheduling algorithm, JobTracker starts to schedule tasks based on the data locality information, available slots on each node. After TaskTracker receives the scheduling requests from JobTracker, TaskTracker regularly sends the heartbeat to JobTracker to report the health status, progress status, and so on. The client application can get the results from

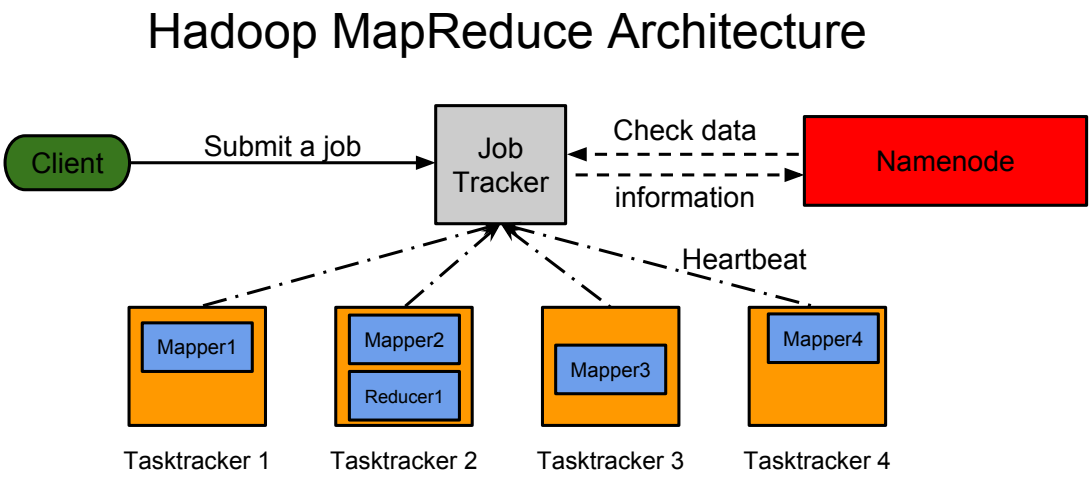


Figure 2.2: Hadoop MapReduce Architecture

the user's appointed output directory in HDFS. JobTracker is also a classical single point of failure in MapReduce. Once JobTracker fails, all running jobs go down. The details can be seen in Figure 2.2.

2.3 Apache Hadoop NextGen MapReduce (Yarn)

To some extent, Hadoop indeed solves the problems in parallel data processing. However, Hadoop jobs are not the only jobs running on the current cluster. Other jobs, such as MPI, shared memory application, and so on, are still running on the cluster. Yarn [15] is proposed to support, manage, and run multiple kinds of applications in the same cluster. In Hadoop, JobTracker takes responsibilities of resource management and job scheduling. This design is too coupling to manage and monitor different kinds of jobs. In order to solve this problem, Yarn separates the duty of JobTracker to two new components: a global ResourceManager (RM) and per-application ApplicationMaster (AM). TaskTracker becomes NodeManager in Yarn. NodeManager manages the resource usage and health status of the node. ResourceManager collects the node information from each NodeManager via heartbeat. ResourceManager includes two main components: Scheduler and ApplicationManager. Scheduler is a pure scheduler. It does not maintain any status of applications in each node.

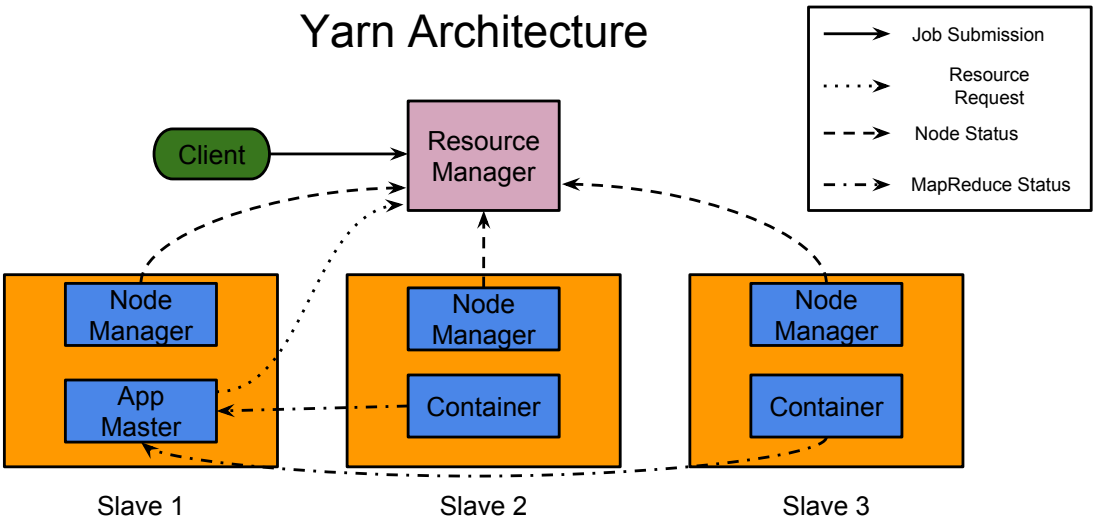


Figure 2.3: Yarn Architecture

ApplicationManager is responsible for track the status of each ApplicationMaster. When a user submit an application to Yarn, according to the resource of the application, ResourceManager first starts an ApplicationMaster for the application. ApplicationMaster negotiates with Scheduler in ResourceManager to start containers. ApplicationMaster collects progress and status information from each container and sends them to ApplicationManager. The details can be seen in Figure 2.3.

2.4 The Hadoop Ecosystem

Around Hadoop, lots of companies involve to develop plenty powerful frameworks. Famous IT companies, such as IBM, Facebook, Twitter, Yahoo, Google, Baidu, and so on. Startup companies concentrated on Cloud Computing, such as Cloudera, Hortonworks, MapR, and so on.

In Figure 2.4, we can find that there are four levels in Hadoop ecosystem. They are application level, access level, processing level, and storage level, respectively. Storage level is responsible for keeping and maintaining the data in cloud. Processing level is used to run parallel processing programs in a specified programming model. In access level, developers

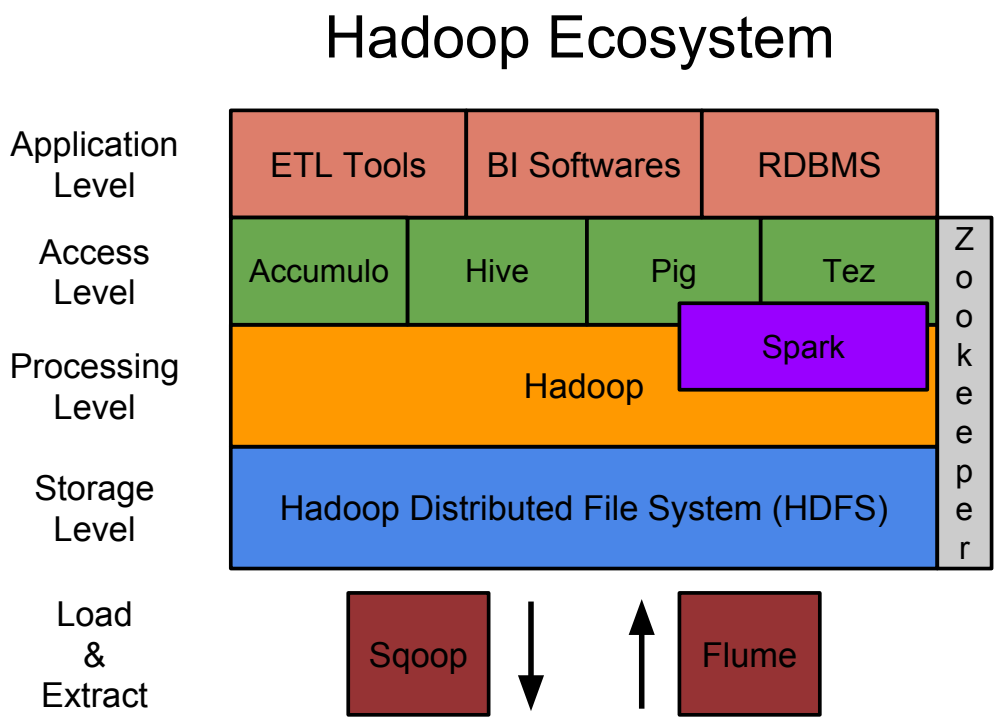


Figure 2.4: Hadoop Ecosystem

and users can directly use the frameworks in this level to implement some concrete and high-level operations on the data. The jobs submitted in access level finally are divided into the different jobs running in the processing level. Application level provides users much easier way to execute the operations they need. Users can focus on their needs without concerning about the implementation inside the cloud environment. Before and after using Hadoop, users should use some tools to help load into and extract from Hadoop storage level. In this area, the famous tools include Sqoop [8], Flume [1], and so on. In the next subsections, we will introduce some famous and important frameworks in the finer classifications including file system, programming model, SQL, and so on.

2.4.1 Distributed File System

Except HDFS, there are serveral other distributed file systems. Red Hat GlusterFS [17] is a scale-out network-attached storage file system. This file system works in Red Hat storage server as a commercially supported integration. Lustre [16] file system is a famous file system

often used in high performance computing environment. Starting from version 2.5, Hadoop supports Lustre file system. Apache Tachyon [10] is an in-memory file system. Tachyon has a very good performance because that the whole file system stores in memory.

2.4.2 Distributed Programming

Some other distributed programming frameworks are proposed. The most famous one is Apache Spark [5]. Spark is an open-source parallel processing frameworks built on top of HDFS. Spark uses a new fault-tolerant method, named Resilient Distributed Dataset (RDD) [50]. With the help of RDD, Spark can recover data very quickly so that Spark keeps a very high fault-tolerance and robustness. Most data in Spark is processed in memory, which reduces the I/O cost. Meanwhile, the design of RDD makes sure that Spark can easily be used to develop an interactive function. Apache Tez [11] is another famous framework. Tez uses an MRR programming model, which means Map-Reduce-Reduce programming model. For each application data, there is only one map phase. For other reduce phases, they can directly use the anterior results.

Spark and Tez natively support directed acyclic graph (DAC). This is very easily to develop some machine learning and data mining programs. And they both have better performance than Hadoop/Yarn.

2.4.3 SQL on Hadoop

Apache Hive [3] is a dataware house framework for processing query jobs built on top of Hadoop. Hive has its own language, called HiveQL. HiveQL is very similar with traditional SQL. Users can learn how to write a HiveQL in a short time. Hive parses the query submitted by the user into multiple Hadoop jobs. Then Hive submits these jobs to Hadoop. After jobs finish, Hive will give results back to the user.

Apache Pig [4] is also a dataware house framework built on top of Hadoop. Pig also has its own language, named Pig Latin. Pig Latin can be used to write data flow, which is a little different from traditional language. Pig has the similar running process like Hive.

Spark SQL [6] (formerly named Shark) is a query processing framework built on top of Spark. The largest difference between Spark SQL and Hive/Pig is Spark SQL parse queries to Spark jobs, not Hadoop jobs. Spark SQL also has a separated query optimization engine, Catalyst. It enables the project has very low coupling. Spark SQL takes Spark advantages so that it has very high throughput.

2.4.4 Data Ingestion

Apache Storm [9] is a parallel processing framework in real time. The frameworks we introduced before runs based on data stored in HDFS or other file system. They cannot handle the real time situation. Storm is developed for solving this problem. Storm has a very general topology so that users can implement the programming model they want. Spark Streaming [7] is a framework on top of Spark. It uses very small batch jobs instead of the completed real time jobs. Spark Streaming uses the similar API of Spark. SO it is very easily used.

We do not introduce all the frameworks listed in the figure because of the limited space. In the next chapter, we will demonstrate the deficiencies in current Hadoop framework and come up with the motivations of our research.

Chapter 3

Motivation

In this chapter, we first introduce more details in the Hadoop Framework. Then we demonstrate the problems of traditional Hadoop MapReduce processing for small files. To motivate this work, we make theoretical analysis to show the reasons of their deficiencies. After that, we use experiments to show the inefficiency and ineffectiveness of Hadoop for processing small files. In the end, we show the research questions and contributions in this research.

3.1 Overview of Hadoop Framework

Apache Hadoop contains three main components: Hadoop Common, HDFS, and Hadoop MapReduce. Hadoop Common provides the necessary utilities that support the other Hadoop modules. HDFS provides scalable, fault-tolerance, and distributed storage service in Hadoop. Hadoop MapReduce is used to parallel processing the large scale of data.

3.1.1 HDFS

HDFS consists of a Namenode and several Datanodes. Namenode takes responsibility of managing all metadata information and responding the file operations. Datanode is responsible for saving the real and replicated data. Each file is split into several blocks with the size of 128MB, which are replicated in HDFS based on the configuration. Namenode regularly receives the heartbeat from Datanodes in a fix time interval, 3 seconds by default. In heartbeat, each Datanode sends its status to Namenode including the Datanode's capacity, used space, remaining space and some other information. After Datanode receives the heartbeat message, it updates each node's status, which is kept in memory. If Namenode cannot get

any heartbeat from one Datanode in 10 minutes, Namenode will remove this Datanode from the live node list and add it into dead node list. Meanwhile, Namenode also stores each file and directory's metadata in the memory. In essence, the consuming memory in Namenode is decided by the number of files stored in HDFS.

3.1.2 Hadoop MapReduce

The first generation of Hadoop struggles in the scalability because of the highly coupled structure itself. The second generation of Hadoop, named Yarn, improves the scalability by seperating Jobtracker into two new components, ResouceManager (RM) and Application-Master (AM). The old TaskTracker has been changed to NodeManager (NM). The AM and NMs use heartbeat to communicate with RM. When a new job is submitted into Hadoop, RM starts an AM on one slave node. The AM sends resource request to the scheduler in RM. After the scheduler allocates the related resource for AM, AM needs to negotiate with NMs to start the containers. Once a container finishes, the AM and NMs can receive the notification and use heartbeat to update information in RM.

3.2 The Issues in Traditional MapReduce for Processing Small Files

In the process of Hadoop MapReduce, Inputsplits are generated by AM. For small files, one file generates one Inputsplit. One Inputsplit is used by one Map container, which means the number of Inputsplits is equal to the number of Map containers. So the first issue is that too many Map containers are created by the MapReduce job for small files. Too many containers mean too many processes. The creating and closing of a process cost very little. However, when there are a large number of processes needed to create and close, the overhead cannot be ignored. The similar overhead is generated for Reduce containers. Another issue is the imbalance between computation ability and parallelism. Excluding the overhead of using processes, the traditional MapReduce processing costs a lot in the disk I/O and network communication. In traditional MapReduce mode, Reduce tasks are used to

gather the different map output files (MOFs) generated by Map tasks, combine the records based on the key, then write the results to HDFS. However, a small file only generates one InputSplit for Map phase, which means only one Map task is launched. In this situation, the role of the Reduce task has been weaken, which means the operations in the Reduce task can be done in the Map task. In this process, the main overhead is generated by storing MOF on the local disk for the Map tasks and extracting MOF via network for Reduce tasks. For a large number of small files, this overhead become heavier.

3.3 Current Hadoop MapReduce Performance for Small Files

In this part, we conduct some tests to collect the MapReduce processing performance in Hadoop. The results show the current Hadoop suffers from processing small files. In the first group of experiments, we compare the performance between a MapReduce job with the input of lots of small files and one with the input of only one single file whose size is equal to the total size of the former files. In the second group of experiments, we run Hadoop jobs with different input sizes. In these experiments, we want to see if different jobs have similar additional periods, which means the period between the point-in-time that we start one job and the point-in-time that MapReduce processing really starts and the period between the point-in-time that MapReduce processing is finished and the point-in-time that the job really finishes[1]. We run these tests on a Hadoop cluster including 1 master node and 8 slave nodes. Each node in the cluster has two 2.67GHz hex-core Intel Xeon X5650 CPUs, 24GB memory and two 500 GB Western Digital SATA hard drives. The configuration in Hadoop keeps the default configuration. The benchmark we choose is famous Wordcount [14] and the testset is created randomly by Linux dictionary file[2]. We prepare ten test sets of small files from 1000 files to 45000 files for the first group of MapReduce tests. The average file

[1]Before a MapReduce job starts the real processing, it needs some time to prepare data and negotiate with AM and RM, we call this period the warm-up period. Similarly, the MapReduce finishing is not equal to the program finishing. We name this period the wind-up period.

[2]Dictionary file (/usr/share/dict/words) is a standard file on all Unix and Unix-like operating systems. It is always used by spell-checking systems, such as ISpell, which includes 235,886 English words.

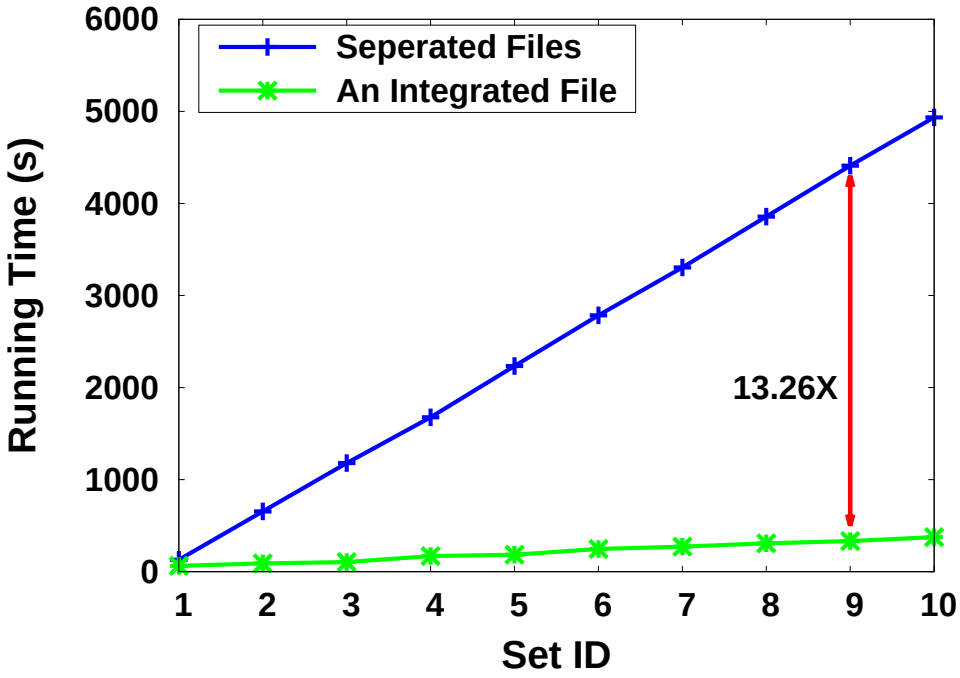
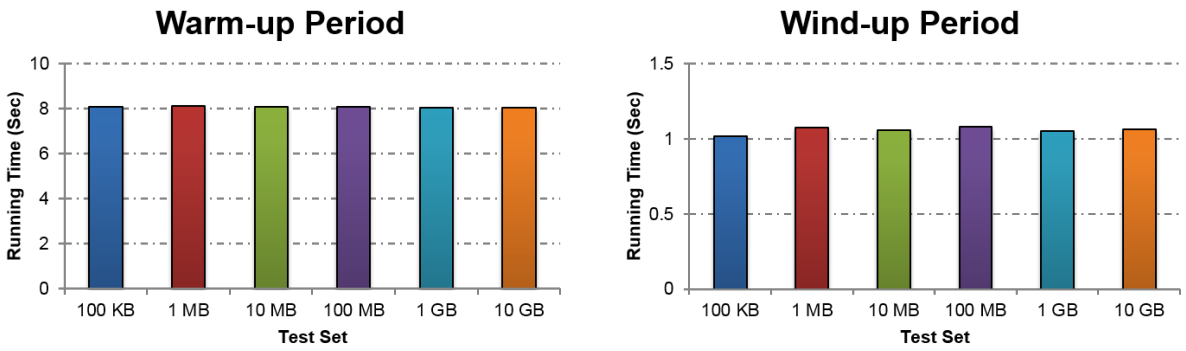


Figure 3.1: Wordcount Performance with Two Different Inputs

size in these testsets is 1023.65 KB, which is a normal size for small files processing [27]. We also integrate these sets of files into ten large files as control groups. For the second group of tests, we create six test files from Linux dictionary file. The sizes of these files are 1 KB, 1 MB, 10MB, 100MB, 1 GB, and 10 GB, respectively.

3.3.1 Hadoop MapReduce Performance Comparison between Small Files and One Integrated File

After storing the ten sets of small files and corresponding integrated large files into HDFS, we try to test the MapReduce performance for these sets. We run original Wordcount provided by Hadoop. There are two input modes in the tests. One is to provide the directory storing all the small files as the input of the Wordcount job. The other is to use one integrated large file mentioned before as the input. We show the experimental results in Figure 3.1. From the Figure 3.1, we can find that with the increase of the number of files, the gap between two input modes grows rapidly. In our small scale testing, the average performance



(a) Warm-up Period Results

(b) Wind-up Period Results

Figure 3.2: Additional Period inside Hadoop MapReduce Processing

of a set of small files is 10.49x lower than the average of integrated file. Especially for set 9, the performance of integrated file is 13.26x higher than the performance of small files. We can deduce that the gap will continue to expand with the increasing number of files.

3.3.2 The Additional Period of MapReduce Jobs with Different Input Sizes

We introduced the definitions of warm-up period and wind-up period in previous sections. Here, we use the same Hadoop cluster and same benchmark, Wordcount, to test the spans of these two period. The size of input data is changed from 1 KB to 10 GB. Shown in Figure 3.2(a) and Figure 3.2(b), the warm-up period and wind-up period changed very little with different sizes. For warm-up period, the running time keeps around 9 seconds. For wind-up period, the runnign time keeps about 1 seconds. This means no matter how large scale of data is used as the input of MapReduce jobs, the warm-up and wind-up periods keeps unchanged.

3.4 Motivation

From the experiments and analysis mentioned above, we can easily and clearly find out that original MapReduce cannot process small files well. With the increasing demands of analytics for small files, It is very meaningful to conclude the best MapReduce strategies

especially for small files. In this paper, we evaluate several different MapReduce strategies for processing small files. We abstract the detailed features from these strategies including original Hadoop, HAR, Accumulo. According to the special characteristics of small files, we optimize the existing MapReduce strategy and name the optimized one SFMapReduce strategy. Compared to original MapReduce, HAR file layout, and Accumulo, SFMapReduce improves MapReduce performance by 14.5x, 20.8x, and 3.83x for different benchmarks on average.

In general, we have made four contributions in this paper:

- We clarify the limitation of the original Hadoop framework for small files, mainly focusing on MapReduce processing.
- We make an analysis of different strategies including Original Hadoop, HAR Hadoop, and Accumulo from the theoretical level.
- We have concluded the best MapReduce strategy, SFMapReduce, in Hadoop from these strategies for running MapReduce programs efficiently for small files.
- We have conducted experiments to test the performance of SFMapReduce strategy compared to the original Hadoop and other strategies. The experimental results show the advantages of SFMapReduce strategy.

In the next chapter, I will assess various MapReduce strategies, conclude the general features of these strategies, and optimize the original MapReduce strategy.

Chapter 4

Assessment of Various MapReduce Strategies For Small Files

In order to solve the issues discussed in Chapter 3, In this chapter, we will first introduce the existing strategies inside and outside Hadoop. We try to compare several mature MapReduce strategies for small files from the theoretical level and conclude some significant and effective characters from these strategies. Then we propose a new MapReduce strategy based on what we learn from these strategies. It should be noted that the discussion and analysis are all about small files.

4.1 Several Existing MapReduce Strategies

As discussed before, Hadoop cannot support well for processing small files. In this section, we focus on demonstrate existing MapReduce strategies. Hadoop provides three strategies, Hadoop Archives (HAR), SequenceFile, and MapFile for small files. BigTable is another MapReduce strategy built on top of Hadoop. After the introduction to these strategies, we show the disadvantages of these strategies from the theoretical level.

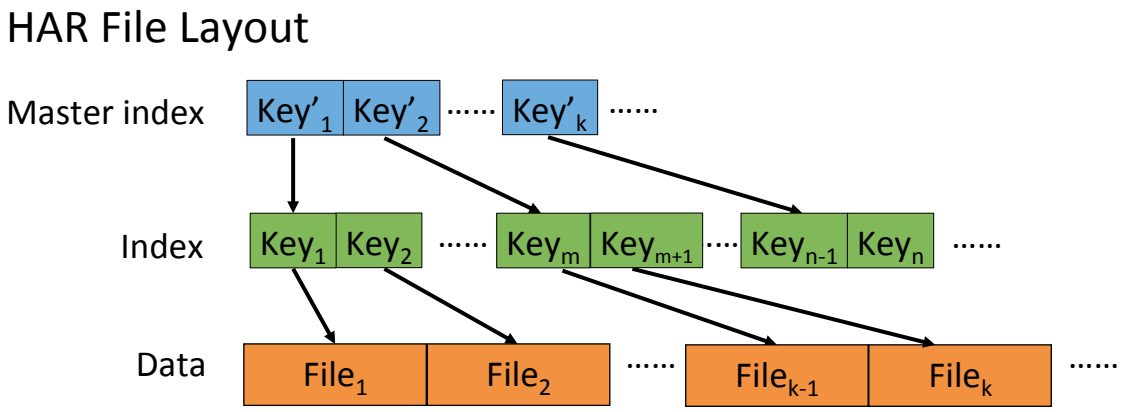


Figure 4.1: Hadoop Archive Layout

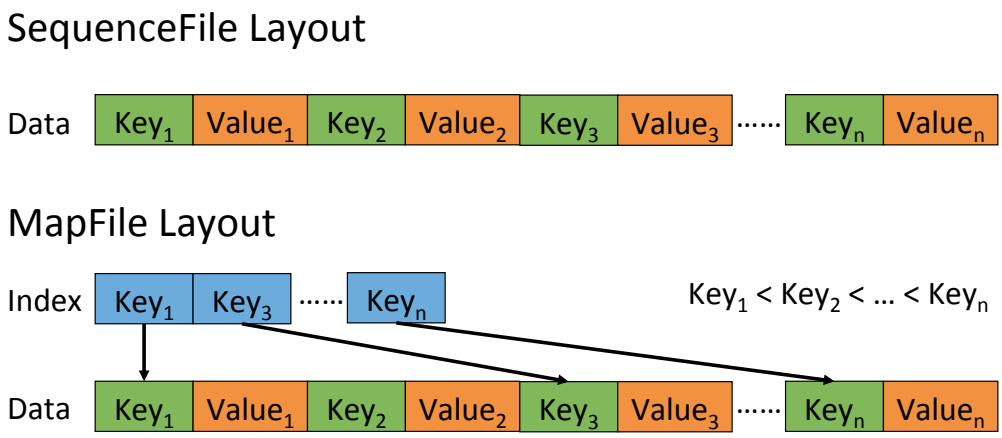


Figure 4.2: SequenceFile Layout and MapFile Layout

4.1.1 HAR Strategy

HAR [26] is a special file layout built in HDFS. It is also a MapReduce strategy for small files. Hadoop MapReduce jobs can use HAR as the input directly. A HAR file usually ends with a .har extension. As shown in Figure 4.1, a HAR file includes a Masterindex, Index and the files data. The Masterindex points to the Index, and the Index includes the file's name and its location information. However, random access supported by HAR seems a little weak. It only supports to random access through the file name.

4.1.2 SequenceFile Strategy

SequenceFile is a very natural file layout and MapReduce strategy. SequenceFile also can be used as the input file format directly. In SequenceFile, keys and values are stored sequentially, as shown in Figure 4.2. We can use file name as the key and file content as the value so that one SequenceFile can be comprised of many input files. Also, this layout can be applied with MapReduce operations directly. However, it cannot provide quick random access to specific files, because SequenceFile does not store the metadata of files. It does not show better performance, compared to files directly stored in HDFS. Like HAR, it only provides one method to select small files based on the key, which is not powerful to meet with users' demands.

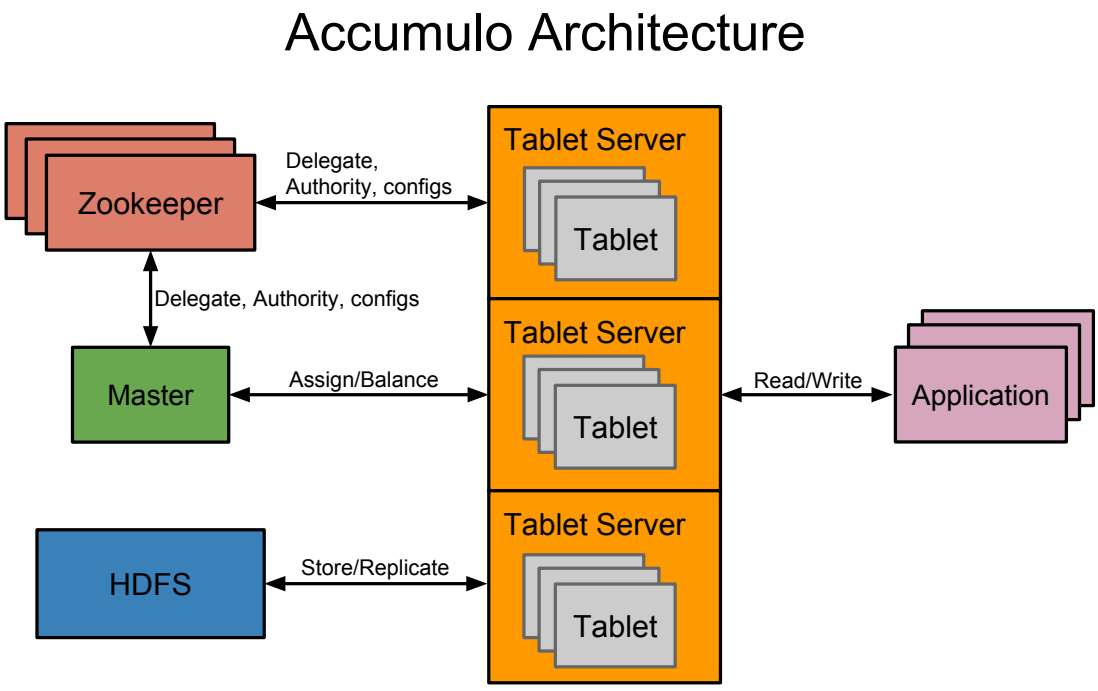


Figure 4.3: Accumulo Architecture

4.1.3 MapFile Strategy

MapFile is an improved layout and strategy based on the SequenceFile, as also shown in Figure 4.2. It consists of two parts, Index and Data. The Key/Value(s) in the data portion are placed in order. Usually, the order is based on the keys. The Index within the MapFile is designed to increase the performance of random access. However, MapFile needs to sort the whole elements before storing them into HDFS and this process costs extra overhead. This means, MapFile needs larger overhead for sorting than the other two strategies.

4.1.4 BigTable Strategies

Except the strategies supported inside Hadoop, users also can try other strategies to improve the MapReduce efficiency. The most convenient and direct method is to use frameworks that provide Key/Value store service called BigTable, such as Accumulo and HBase. Here, we choose Accumulo as the strategy. Accumulo is chosen as it has very impressive

proven I/O performance, outperforming Oracle, Cassandra or Hbase [34] with 100M ingests/second while keeping comparable querying speed [40]. Besides, it's also for cell-level security option that is not offered in other peers and was used for several high-performance cases. It was also used for several high-performance applications dealing with huge load of small files such as network events analytics [39]. Accumulo is a distributed Key/Value store framework based on the "Big Table" technology from Google. This system is built on top of Hadoop, Zookeeper, and Apache Thrift. Accumulo provides a very richer data model than simple Key/Value stores. In the key part, there are some other attributes like RowID, Column, Timestamp, and Value. In Column, users can store more details for the Key/Value pair. Accumulo includes many TabletServers, one Garbage Collector process, one Master server and many Clients. TabletServers are responsible for managing some subsets of all the tablets stored in Accumulo like writing, persisting, and flushing. Garbage Collector periodically identifies the files that are not needed any more and deletes them. Master and Clients together take the responsibility of availability, robustness, and recovery. The general processes in Accumulo can be seen in Figure 4.3.

With Accumulo, users can set a Key/Value pair for each small file. The key includes UUID, file name, and some other important information. The value is the file content. Users can directly use the related services provided by the frameworks to improve the performance of storage and computation.

4.2 What We Can Learn from Existing Strategies

We have introduced several MapReduce strategies in the last section. In this part, we make an theoretical analysis of different strategies. After showing their deficiencies, we conclude the features from these MapReduce strategies for small files.

4.2.1 The Deficiencies of Existing MapReduce Strategies

HAR Strategy: The disadvantage of HAR strategy is that HAR file does not support any optimizations of MapReduce. It still generates a large number of InputSplits. In addition, in the process of MapReduce, more I/O overhead is imported to parse HAR files and extract KV pairs. The running of MapReduce jobs with HAR strategy costs more time than the input files stored directly in HDFS. It default uses traditional MapReduce jobs so that there are no optimizations in MapReduce programs.

SequenceFile Strategy: SequenceFile has the similar performance with HAR strategy because it has no optimizations of MapReduce. What's more, it does not have an index so that SequenceFile strategy cannot provide a very flexible MapReduce approach to users. It has the same situation that there are no optimizations in MapReduce programs.

MapFile Strategy: MapFile does not try to provide any optimizations for the MapReduce processing. It runs the MapReduce jobs following the original way in Hadoop, so it has the similar performance with HAR strategy. It is more flexible than SequenceFile because of the index part in the MapFile. It default runs MapReduce job.

Accumulo Strategy: The objective of Accumulo is not specific to small files. Actually, their management services provide too general operations, which hurt the performance to some extent. For example, Accumulo has to run more processes to make the framework more security. Another example is that Accumulo automatcally compress the data stored in it, which means when we write the results of MapReduce jobs into Accumulo, the compression overhead will impact on the MapReduce performance. A good design in Accumulo is the combination of input files for a MapReduce jobs. It makes sure that Accumulo does not generate a large number of InputSplits. The processing performance is good but still has a little more overhead. For the MapReduce programs, Accumulo does not give any recommendations. More detailed analysis can be seen in Chapter 5.

Table 4.1: Feature Summary

Strategy	Efficiency	Functionality	Additional Overhead
Original Hadoop	Low	Low	Low
HAR	Low	Middle	Middle
Accumulo	Middle	High	High
SFmapReduce	High	High	Low

4.2.2 Summary of General Features from MapReduce Strategies

From the analysis above, we can extract three important general features for MapReduce strategies. They are efficiency, functionality, and additional overhead. Efficiency presents the MapReduce running efficiency for small files. In efficiency, we consider each detail in MapReduce processing. Functionality shows whether the strategy can provide users powerful MapReduce processing. Additional Overhead is another very important indicator for MapReduce strategies. Although some strategies can reach the good efficiency, they bring in much overhead. Then we cannot consider that this kind of strategies is very good. We use these features to evaluate the performance of MapReduce strategies. Table 4.1 shows the efficiency and functionality of Original Hadoop, HAR[1], and Accumulo.

Original Hadoop and HAR strategies have low efficiency. Because they treat small files equally as large files, which leads to a large number of InputSplits. This operation hurts the performance of MapReduce. Accumulo can combine small files into one InputSplit. However, the size of InputSplit is fixed at 2 GB without any changes for different situations. So Accumulo's efficiency is middle. These three strategies use traditional MapReduce jobs, which include Map tasks and Reduce tasks. From this perspective, MapReduce jobs are not optimized in these strategies so their efficiency is low. Looking at the above factors, we can know the efficiency of Original Hadoop, HAR, and Accumulo are low, low, and middle. For functionality, original Hadoop and HAR cannot provide any customized operations for users so their functionality is low. Accumulo can meet with users demands in the prepartaion

[1]HAR, SequenceFile, and MapFile are very similar. Here, we choose the most poplular one, HAR, to analyze.

time before starting MapReduce processing, so Accumulo's functionality is high. Although Accumulo has very good efficiency and functionality, its additional overhead should be considered. Accumulo compresses data before storing data into it and it is compulsive to wait for the copying completion of current file's replica before starting to store the next file. What's more, for the security reason, Accumulo starts some special processes, which reduces the performance. So Accumulo's additional overhead is high. HAR strategy needs a little overhead when the MapReduce job extracts KV pairs from the special HAR file layout. The additional overhead of HAR is middle. Compared to the two former strategies, original Hadoop does not incur additional overhead so its overhead is low.

4.3 MapReduce Strategy Optimization for Fast Analytics of Small Files

From our analysis, existing MapReduce strategies is not suitable for small files because they are designed for large files. It is necessary for us to design a new MapReduce strategy for small files. As we have extracted crucial features for designing a good MapReduce strategy, we can optimize the MapReduce strategy especially for small files. For simplicity, we name our new strategy small file MapReduce strategy (SFMapReduce). In the following paragraphs, we mainly demonstrate the details of the new strategy we designed from the perspectives of efficiency, flexbility, and additional overhead.

4.3.1 Efficiency

In this section, We will talk about the optimized details in the efficiency feature. The main topics start from General Efficiency and MapReduce Efficiency.

Optimization of MapReduce Components: In Hadoop, InputSplit is a minimum input unit for the map task. The information kept by InputSplit includes file path, the data offset in the file, the data length in the length, the list of nodes where the file is stored, and the information of related storage blocks. The number of InputSplits directly decides the number of map tasks. Usually, the large file used in MapReduce programs produces more

than one InputSplit. However, for the target small files, Hadoop normally generates one InputSplit for one file. This means Hadoop has to create a large number of map tasks, *i.e.* processes. To create and close lots of processes and context switching frequently incur a lot of overhead, which really harms the performance.

RecordReader is another very important component in Hadoop MapReduce. RecordReader is responsible for transferring the byte stream, provided by InputSplit, to the record stream (KV pairs), used by map tasks. However, the original RecordReader cannot support our optimization well because the elements in KV pairs have be changed. In order to solve this problem, we should optimize MapReduce components and related algorithms for achieving the better efficiency.

In order to decrease the number of InputSplits, we choose CombineFileSplit as the InputSplit of map tasks. CombineFileSplit is a built-in InputSplit in Hadoop, which is a set of input files. In Hadoop, InputFormat is used to generate InputSplits. InputFormat describes the input-specification for a MapReduce job. For creating CombineFileSplits correctly and decreasing the number of map tasks, we design SFInputFormat as a component in our new strategy. SFInputFormat is used to parse and collect the information of small files stored in HDFS and generates a list of CombineFileSplits. For the new RecordReader, SFRecordReader, we override the related functions to provide the necessary services to map tasks. According to the requests from map tasks, SFRecordReader parses the input list of CmobineFileSplits and returns indentification of one file as the Key and corresponding file data as the Value.

Optimization of Partition Algorithm: The partition algorithm in SFInputFormat is the most significant because it is used to generate the CombineFileSplits. A very important threshold, named SFSplitSize, is used to limit the total size of the files contained by a CombineFileSplit, which decides how much data one InputSplit can contain. Our partition algorithm shows its details in Algorithm 1. First, the algorithm first checks if we have initialized the CombineFileSplit. If it is existed, then continue running; if not, initialize a new

Algorithm 1 Partition Algorithm

```
1: Input: SFInfo: sfInfo, the threshold of split size (SFSplitSize): size_limit
2: Output: A List of InputSplit: lis
3: Initialization
4: while sfInfo.hasNext() do
5:    if InputSplit is = NULL then
6:       is ← NewCombineFileSplit()
7:       is.size ← 0
8:    end if
9:    SFIndex sfIndex ← sfInfo.getIndex()
10:   if Selector(sfindex) then
11:      Add the file path, offset, length, and block information into the CombineFileSplit
         is
12:      is.size += sfIndex.getLength()
13:      if is.size > size_limit then
14:         lis.add(is)
15:         is ← null
16:      end if
17:   end if
18: end while
19: if is ! = NULL then
20:   lis.add(is)
21: end if
```

CombineFileSplit (lines 5-8). The next step is to get the index information of all files (line 9). After getting the SFIndex, we first use *s*elector() function to check whether this file meets with the users' requirements. (line 10). If the result is true, then we extract the necessary information and store it in CombineFileSplit (line 11). Then we update the size of current CombineFileSplit (line 12) and continue to check if the size reaches the limit of SFSplitSize (lines 13-16). If the size is more than the SFSplitSize, then we add the CombineFileSplit into the list and clear the InputSplit variable. Finally, the function returns the list of CombineFileSplits. From the algorithm, we can find out that SFInputFormat creates a much less number of InputSplits than traditional way in Haddop for map tasks. This can decrease the number of map tasks, which improves the performance of our framework in turn.

Optimization of Input Size (SFSplitSize): In our new MapReduce strategy, the input can include a lot of small files. So the SFSplitSize is very important. It directly affects the performance of MapReduce. For the best performance, we should make a balance between data access overhead and parallelism. If the SFSplitSize is too small, the parallelism is degraded; if the SFSplitSize is too large, the data access overhead is increased; this size is very hard to decide because of the complexity of MapReduce processing. In the next chapter, we conduct multiple experiments to choose the best SFSplitSize.

Optimization of KV Pairs: In original MapReduce, KV pairs are used in a very detailed way. For example, for Wordcount benchmark, in the map output file, the Key is one word appearing in the text file and the Value is 1. It is very intuitive and efficient for computing and collecting results for large files. However, for small files, we should use Key as the identification of one file and Value as the data of the file. With this optimization, one KV pair corresponds to one file.

Optimization of Program – Map-Only Progarm: As discussed above, only one map task is created for small job. Obviously, map task can directly compute the final results without any help from others. So that is why there is only Map function existing in our SFMapReduce application. More specifically, in the traditional MapReduce applications, the map task costs I/O and computing overhead when storing map output file (MOF) into local disk. The reduce task degrades the performance because of the extra cost of creating process and network communication. In our Map-Only program [2], we start the write input stream for the destination file in *setup*() function and close it in *cleanup*() function inside the map class, which helps us finish the work previously running in reduce tasks. In this process, we use HDFS APIs to write the results to the destination path in HDFS.

[2]Map-Only program is our recommendation for processing small files. Actually, SFMapReduce strategy can support all kinds of MapReduce programs.

4.3.2 Functionality

For the functionality and flexibility, we design a *selector* component in the SFInputFormat. To be specific, we add a *selector()* function in the SFInputFormat class. According to the SFIndex in the input file, the *selector()* function can easily get the information of each small file. Before adding a small file into the InputSplit, we need first consider the return value of *selector()* function. If the file's information accords with the conditions, then *selector()* returns true and this file is added into the InputSplit; If not, returns false and this file is skipped. With the help of the *selector()* function, users can run the jobs flexibly and selectively. In our framework, default *selector()* function always returns true. Users can write any complex judgments inside the code based on their demands. This optimization provides good functionality and flexibility to users.

4.3.3 Additional Overhead

In our new MapReduce strategies, we should reduce the additional overhead, especially on I/O and communication. Accumulo has the largest overhead from extra compression, synchronization of replicas, to the storing balance. We should avoid this kind of overhead in the new strategy. HAR automatically combines small files into several 512 MB DFS blocks when users add files into HAR. This operation incurs remote copying of blocks a lot when running MapReduce jobs. The simpler, the more effective. We try our best to follow the easiest method, which is very similar with original MapReduce. The additional overhead in our strategy is only some I/O write operations for writing results into HDFS. This shows the additional overhead in SFMapReduce is very low.

4.4 Implementation

We have implemented our framework based on Hadoop/Yarn 2.6. We finish the SFInputFormat and SFRecordReader class derived from existing classes in Hadoop and override

the related functions. The related partition algorithm is implemented in the SFInputFormat class. We also implement multiple Map-Only applications for experiments in the next chapter.

Chapter 5

Evaluation

In this section, we evaluate the effectiveness of SFMapReduce strategy compared to existing strategies. We first describe the experimental environment, then show the experimental results with the analysis.

5.1 Experimental Environment

5.1.1 Cluster Setup

Our private cloud consists of 17 computer servers, each with a 2.67 GHz hex-core Intel Xeon X5650 CPU, 24 GB memory and two 500 GB Western Digital SATA hard drives. The machines are connected through 1 Gigabit Ethernet. In experiments, we create a Hadoop cluster of 17 nodes.

5.1.2 Hadoop Setup

We use Hadoop/Yarn-v2.6.0 as the code base with JDK 1.7. One node is dedicated as the ResourceManager and the Namenode of Yarn and HDFS. In our cluster, we have 16 slave nodes. The key configurations of Hadoop/Yarn we use are shown in Table 5.1.

5.1.3 Benchmark

In our experiments, we choose the most representative program, Wordcount, TeraSort, and Grep to test the performance. It should be noted that the these three benchmarks are Map-Only applications. The reason why we use Map-Only program has been discussed in the Section 4.3.1. The details of these benchmarks can be seen in [14], [13], and [47].

Table 5.1: List of key Hadoop configuration parameters.

Parameter Name	**Value**
yarn.nodemanager.resource.memory-mb	22528 MB
yarn.scheduler.maximum-allocation-mb	6144 MB
yarn.scheduler.minimum-allocation-mb	2048 MB
yarn.nodemanager.vmem-pmem-ratio	2.1
MapReduce.map.java.opts	2048 MB
MapReduce.reduce.java.opts	2048 MB
MapReduce.task.io.sort.factor	100
dfs.block.size	128 MB
dfs.replication	3
io.file.buffer.size	8 MB

5.1.4 Alternative Strategies Setup

Except comparing to original Hadoop, we also compare SFMapReduce with one strategy inside Hadoop (HAR) and BigTable strategy outside Hadoop (Accumulo).

HAR Strategy: We have introduced HAR strategy in the last chapter. In our test, we use the data prepared to create the corresponding HAR files, then use these files as the input of related benchmarks.

Accumulo Strategy: Accumulo strategy is also mentioned in the last chapter. Accumulo is chosen as it has very impressive proven I/O performance [34], high throughput [40] and cel-level security [39] in our tests. We use the latest Accumulo version 1.61 as one control group. According to the our experience, for the best performance, the number of tablet servers should be set to about 40% of total nodes. So we randomly pick 5 tablet servers from a 17-node cluster. The tuple stored in Accumulo includes the key, user defined information (file name, create time, and so on), and the value (file content).

5.1.5 Data Preparation

For Wordcount and Grep, we randomly extract part of Shakespeare Complete Works [18] to create lots of files with different sizes. For TeraSort, we use TeraGen to randomly create the number of lines in the result. After preparing the data, we divide the small files into 4

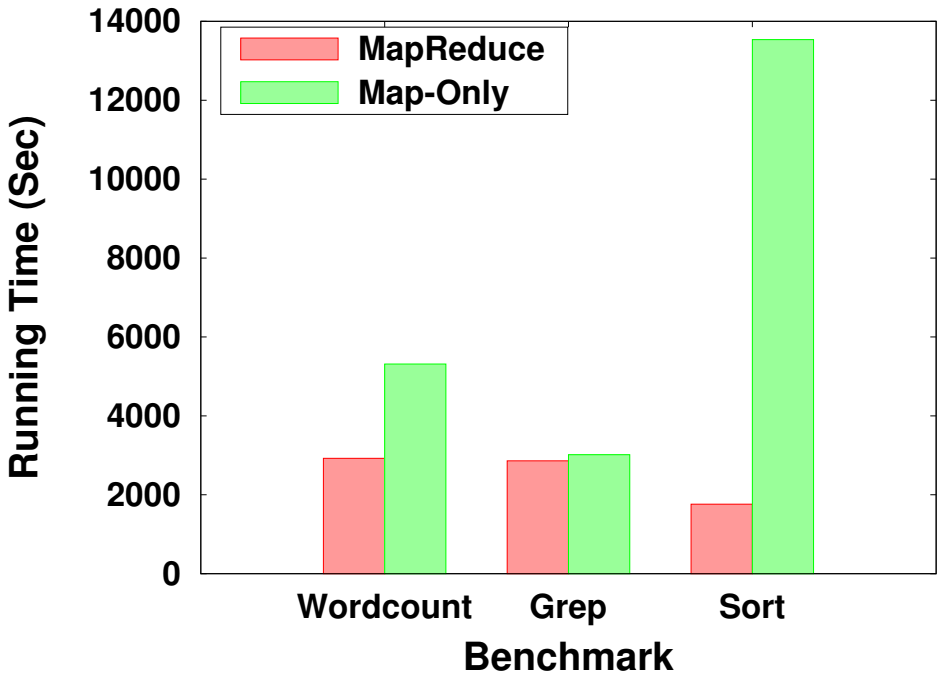


Figure 5.1: Performance of Map-Only and MapReduce jobs

sets for each benchmark. The total sizes of the sets are 10 GB, 20 GB, 40 GB, and 60 GB, respectively.

5.2 Map-Only Application Performance

We start our experiments to prove Map-Only Application can provide better performance than original MapReduce Application, discussed in Section III-C. For comprehensiveness, we use the three benchmarks for testing. We change slightly MapReduce programs to Map-Only programs for these benchmarks. The input size of these benchmarks is all 60 GB. For stability, we run each test 5 times and collect the average running time. As shown in Figure 5.1, the average running time of Map-Only jobs is less than original MapReduce jobs by 3.52x. Specifically, Map-Only jobs have 1.81x, 1.05x, 7.68x better performance than original MapReduce jobs, for benchmark Wordcount, Grep, and Sort. From the results, we can find that the more network communication a MapReduce job need, the worse performance it has. The reason is the benefit of Map-Only jobs comes from avoiding the overhead from

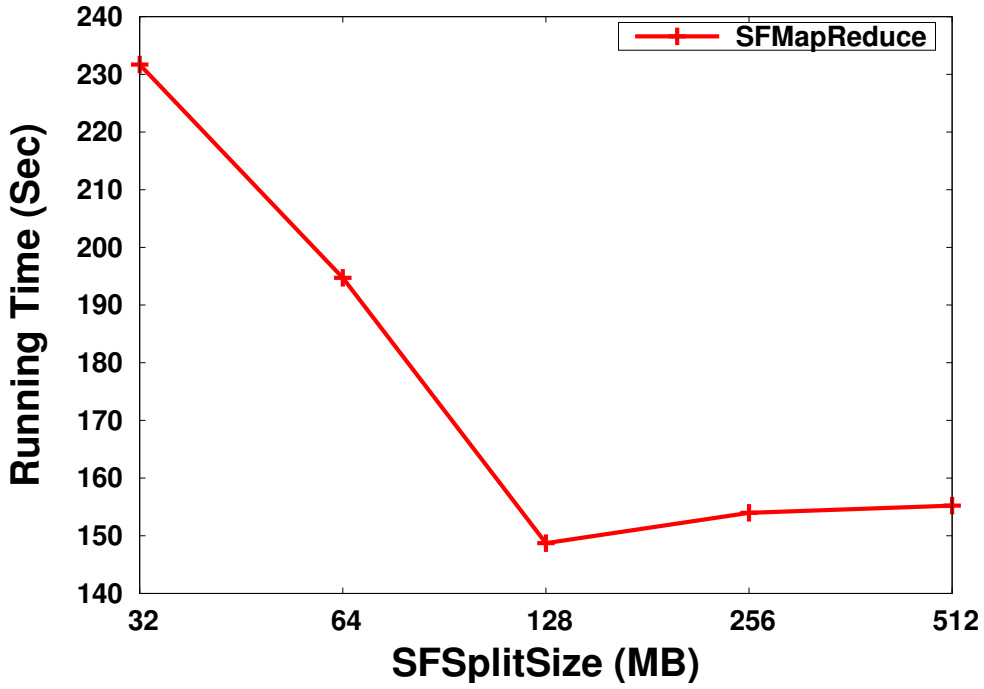


Figure 5.2: The Impact of SFSplitSize

the intermediate data and reduce phase. If the intermediate data of MapReduce jobs is very small, like Grep job, the performance between these two kinds of jobs is very similar. Based on these experiments, we choose Map-Only jobs instead of traditional MapReduce jobs in SFMapReduce. Although we choose Map-Only jobs in SFMapReduce, it does not show the incapability of MapReduce jobs in our strategy. If users need to run MapReduce jobs for some special condition, SFMapReduce can perform it very well. In the rest parts of this paper, we use Map-Only jobs to replace the original MapReduce jobs.

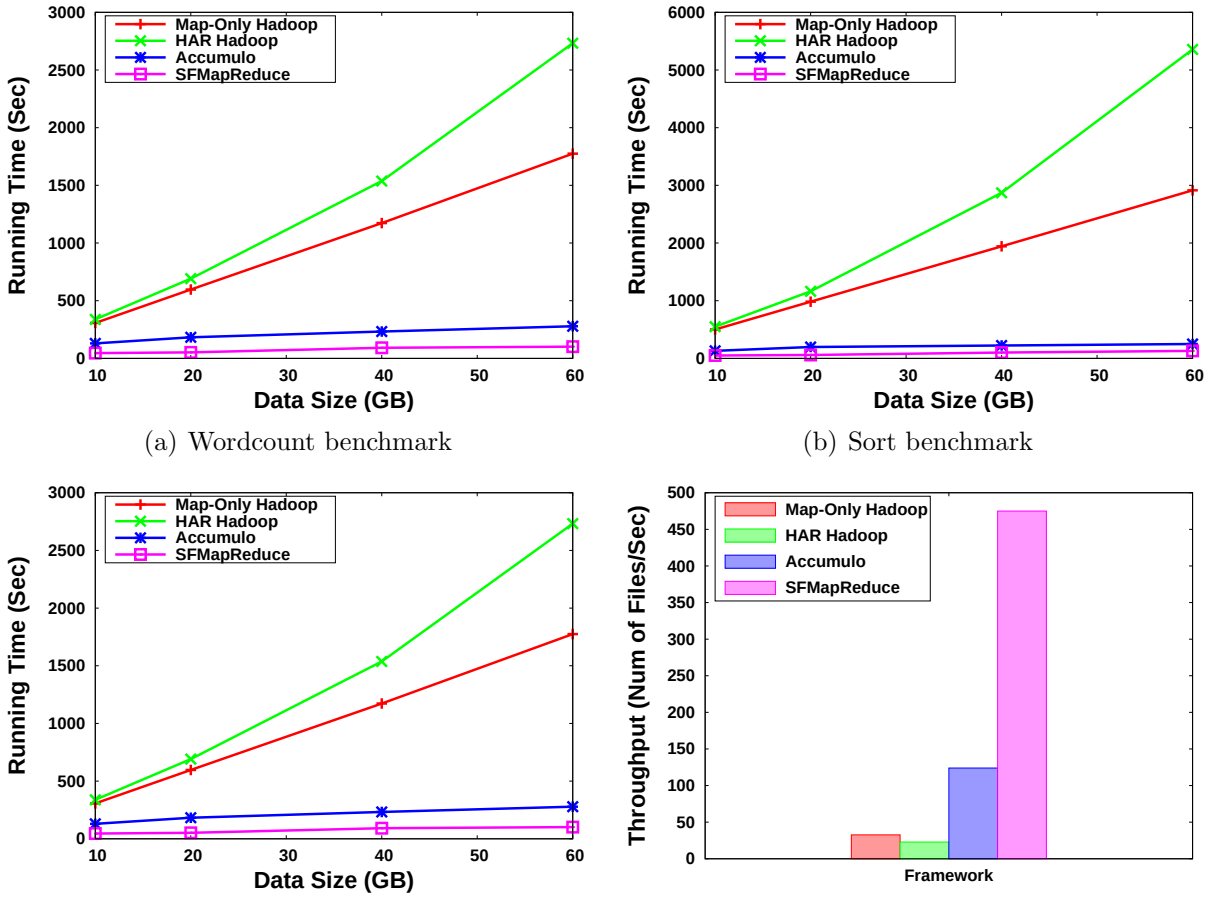
5.3 Tuning the Size of SFSplitSize

In this section, in order to get the best performance of SFMapReduce, we want to tune the size of SFSplitSize, which directly decides the size of an InputSplit. For split size, we should make a balance between data access and parallelism. We use Wordcount benchmark to test SFMapReduce performance with different sizes of SFSplitSize from 32 MB to 512 MB. The results are shown in Figure 5.2. From the results, it is obvious that when the value

of SFSplitSize is equal to 128 MB, SFMapReduce performs best. Especially, in our strategy, HDFS block size is 128 MB. This means when the SFSplitSize is near or equal to HDFS block size, we can achieve the best performance. If the value of SFSplitSize is too small, then data access overhead is increased; if it is too large, the task processing parallism is degraded. It is why when the size of SFSplitSize is near or equal to HDFS block size, SFMapReduce performs best. In the rest part of evaluation, we choose 128 MB as the value of SFSplitSize.

5.4 Overall Performance

In this section, we test the SFMapReduce processing performance, compared to different strategies. Inside SFMapReduce, we set the SFSplitSize to 128 MB based on the tuning results shown in the previous section. The scales of data set are 10 GB, 20 GB, 40 GB, and 60 GB, respectively. We conduct the experiments 5 times and collect the average running time. The test strategies include Original MapReduce Hadoop, HAR Hadoop, Accumulo, and SFMapReduce. In these strategies, we all use Map-Only Wordcount, Sort, and Grep benchmark. Map-Only Hadoop means the original Hadoop with Map-Only jobs. HAR Hadoop means the Hadoop processing with the HAR layout. As shown in the figure 5.3, on average, the performance of SFMapReduce overperforms Map-Only Hadoop by 14.5x, HAR Hadoop by 20.8x, and Accumulo by 3.8x. For Wordcount benchmark, the performance of SFMapReduce overperforms Map-Only Hadoop by 14.4x, HAR Hadoop by 21.0x, and Accumulo by 6.1x; for Sort, SFMapReduce is better than Map-Only Hadoop by 12.1x, HAR Hadoop by 16.1x, and Accumulo by 2.9x; for Grep, our strategy runs faster than Map-Only Hadoop by 17.1x, HAR Hadoop by 25.2x, and Accumulo by 2.5x; Figure 5.3(d) shows the efficiency of our strategy from the perspective of processing throughput on average. Original Hadoop and HAR Hadoop don't make any optimizations in the MapReduce processing. Accumulo uses more general solutions, which are not special to small files and bring in other overhead. SFMapReduce provides best performance because it reaches the three main features we concluded in the last chapter. High efficiency helps improve the MapReduce



(a) Wordcount benchmark

(b) Sort benchmark

(c) Grep benchmark

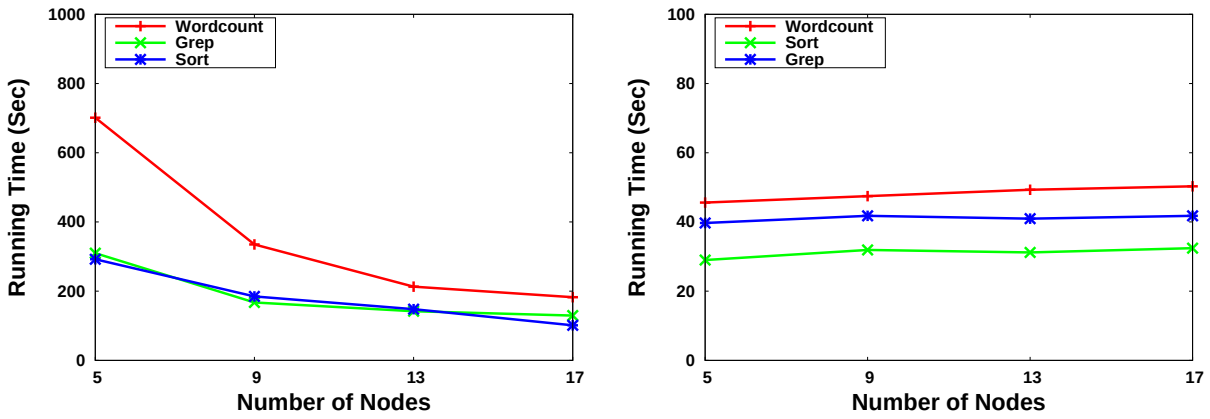
(d) File Throughput

Figure 5.3: Job Running Time

performance. High flexibility provides users autonomy for running MapReduce jobs. Low overhead further increases the performance. The whole processes are direct and highly efficient.

5.5 Scalability

After talking about the overall performance of SFMapReduce, we continue to show the scalability of SFMapReduce. There are two kinds of scalability. One is strong scaling, which means the performance of the strategy running on an increasing number of computation resources, i.e. nodes. If a strategy has a good strong scaling, the performance should become better with the increasing number of nodes. The other is weak scaling. Weak scaling shows



(a) SFMapReduce Scalability with Increasing Number of Nodes

(b) SFMapReduce Scalability with Increasing Data Size

Figure 5.4: SFMapReduce Scalability

when the input data is not so large, the efficiency of the strategy on an increasing number of nodes. If a strategy has a good weak scaling, the performance should keep similar to the increasing number of nodes. In experiments, we first to test the strong scaling with the different data sets on the clusters of 5 nodes, 9 nodes, 13 nodes, and 17 nodes, respectively. Then we test the weak scaling of SFMapReduce on the clusters of 5 nodes, 9 nodes, 13 nodes, and 17 nodes with three benchmarks, where each node deals with 1 GB data. For each results, we collect the average running time based on 5 times experiments results. As shown in Figure 5.4(a), in average, SFMapReduce on the cluster of 17 nodes can keep an improvement of 2.93x over the cluster of 9 nodes and 1.76x over the cluster of 5 nodes. These results show the strong scaling of SFMapReduce. In Figure 5.4(b), we can see, the performances on different clusters keep unchanged. This proves that SFMapReduce has very good weak scalability.

Chapter 6

Related Work

In this chapter, we demonstrate the important and innovative researches closely related to our research. In the first part, we want to introduce some researches conducted to solving the small files problem in HDFS. After that, we elaborate the researches about how to improve the Hadoop MapReduce performance.

6.1 Handling Small Files in HDFS

Chen et al. [22] proposes two techniques to solve the problem of small files in HDFS. The first technique is very similar with our SFLayout. They integrate the small files into a large file and build an index for them. The second technique is to build a cache level between HDFS and users. This cache can help improve the performance of HDFS read and write. However, this paper only focuses on how to improve the I/O performance of HDFS. They do not give a solution about how to use it effectively in the MapReduce processing.

Zhang et al. [51] introduce a very similar research with [22]. The authors build a non-blocking I/O to merge small files into a large file in HDFS. However, they do not consider how to process the data stored in HDFS, either.

Mackey et al. [37] use HAR file layout to solve the small files problem. They focus on how to schedule jobs in Hadoop cluster with quota policy. HAR file layout is not a flexible and powerful layout. They do not provide good solutions for consecutive MapReduce processing.

Dong et al. [26] propose a case study paper researching on small files problem in HDFS. This paper is not only a case study paper but also a research including many techniques and related discussions. In this paper, the authors provide a technique to integrate small files into a large file in HDFS. It is very similar with techniques from other papers. In addition,

the authors provide a prefetching technique to support the function needed in their real education system. Research [28] is the authors' subsequent research. In this paper, the authors divide the files into three categories: structurally-related files, logically-related files, and independent files. For the first two types, they design the special storing solutions. They also introduce a three-level prefetching and caching strategies. However, both of the papers lose the consideration about how to run MapReduce jobs in the new Layout. They only focus on how to meet with the demands of a concrete system.

In [32] and [20], the authors propose very similar solutions with the researches introduced above. They both do not take the MapReduce processing into account.

6.2 Hadoop Improvement

In order to improve Hadoop comprehensively, a large number of researches have been done. They have optimized Hadoop from the following several main aspects. The first aspect is to improve the interaction of Hadoop. The second main area is to improve the performance of Hadoop. The third side is to increase the fault tolerance and application scenarios of Hadoop.

To improve the interaction of Hadoop, many researchers make great efforts on it. The most famous one is MapReduce Online [24]. The authors break the block in task level and job level. They change the processing model from batch model to pipeline model. This model help the final job can get the data in advance, which the final job can return users early inaccurate results. The early results can help users make flexible and meaningful decision. Laptev et al. [35] propose a framework based on Hadoop providing early results. They choose different way from MapReduce Online. They do not change the original processing model in Hadoop. They use heuristic method to try to provide the satisfied results to users. In the beginning, the framework only tries to run regular jobs on a small set. If the results cannot meet with the demands of users, the framework will use a larger set to run the related jobs.

The framework run this steps until the results can meet with the demands of users. These two researches really expand the interaction and functionality of Hadoop framework.

There are plenty of papers done on how to improve the performance of Hadoop. Wang et al. [44] propose Hadoop-A, an acceleration framework that optimizes Hadoop with plugin components implemented in C++. Originally, Hadoop starts reduce tasks after the finished map tasks exceed the preset proportion. Most reduce tasks cannot finish their jobs because they have to wait for finishes of all the map tasks. In Hadoop-A, the framework delay starting the reduce tasks and store some map output information in advance. So when all the map tasks finish, the reduce tasks run merge and reduce processing in pipeline, which improves the performance a lot. What's more, Hadoop-A uses Remote Direct Memory Access (RDMA) to replace the traditional in communication level. This technique also helps to reduce the latency of network in the framework. Li et al. [36] propose CooMR, a cross-task coordination for data management in Hadoop. In this paper, the authors use cross-task opportunity memory sharing and log-structured I/O consolidation to implement cross-task coordination. They also provide key-based in-situ merge to reduce the cost of merge processing. Wang et al. [45] use experiments to prove the cost of data shuffling and merging in Hadoop. The reason is the related operations bulit on top of JVM. The authors further use C++ implementation to replace the JVM implementation existing in Hadoop. The results show this can reduce the execution time by up to 66.3% and lower the CPU utilization by 48.1%.

Wang et al. [43] propose two techniques: analytics logging (AL) and speculative fast migration (SFM) to eliminate failure amplification and fast job recovery execution. AL is a lightweight technique that stores the progress logs of MapReduce tasks. SFM solves node failures by re-executing map tasks, move reduce tasks, and merging with a pipeline of shuffle/merge and reduce stages. This work is so effective to reduce the recovery cost of reduce tasks failure. Xu et al. [49] introduce a new Virtual Analytics Shipping (VAS) framework through integrating MapReduce programs with Lustre storage system. This

research expands the range of adaptability for MapReduce. In high performance computing environment, users can easily use MapReduce to process the data storing in Lustre for a long term.

Chapter 7

Conclusion and Future Work

Hadoop is a powerful and widely used framework to handle large scale of data. Users and developers can easily use Hadoop to parallel process the data in an available and scalable cloud environment. The demands and requirements vary dramatically in the practical world. One of the most significant demands is to add features to support processing small files in Hadoop. As we discussed in motivation chapter, MapReduce in the original Hadoop cannot support small files well. In order to solve these problems, we assess the various existing MapReduce strategies and conclude three important features from them. According to what we observe and learn about, we further optimize the MapReduce strategy and name it SFMapReduce strategy. In SFMapReduce strategy, we optimize the efficiency, flexible, and overhead for MapReduce processing. Our experiments show that SFMapReduce can provide very efficiency, flexible, and scalable MapReduce processing.

There is still a lot of work needed to do for SFMapReduce. In my opinion, I show several research topics that I think are the most important and interesting ones. The most interesting me is whether we can exploit other devices to future improve the performance. For example, if we can integrate the powerful computation ability of GPU into our work, then we can get better performance. The second one is how to solve the data streaming for small files. The data is generated at anytime. So if we can extract the data as long as it is created, we can save a lot of time and even provide the immediate results to users. The last one is maybe we can add pipeline mode in map processing stage and storing stage in map task. I think these three topics are our future research directions.

Bibliography

[1] Apache Flume. http://flume.apache.org/.

[2] Apache Hadoop. http://hadoop.apache.org/.

[3] Apache Hive. https://hive.apache.org/.

[4] Apache Pig. http://pig.apache.org/.

[5] Apache Saprk. http://spark.apache.org/.

[6] Apache Spark SQL. https://spark.apache.org/sql/.

[7] Apache Spark Streaming. https://spark.apache.org/streaming/.

[8] Apache Sqoop. http://sqoop.apache.org/.

[9] Apache Storm. https://storm.apache.org/.

[10] Apache Tachyon. http://tachyon-project.org/.

[11] Apache Tez. http://tez.apache.org/.

[12] Apply new analytics tools to reveal new opportunities. http://www.ibm.com/smarterplanet/us/en/business_analytics/article/it_business_intelligence.html.

[13] Hadoop Grep Wiki. http://wiki.apache.org/hadoop/Grep.

[14] Hadoop Wordcount WIKI. http://wiki.apache.org/hadoop/WordCount.

[15] Hadoop Yarn. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[16] Lustre File System. http://lustre.org/.

[17] Red Hat GlusterFS. http://www.gluster.org/.

[18] Shakespeare Complete Works. http://www.gutenberg.org/ebooks/100.

[19] M. H. Almeer. Hadoop mapreduce for remote sensing image analysis. *International Journal of Emerging Technology and Advanced Engineering*, 2(4):443–451, 2012.

[20] S. Chandrasekar, R. Dakshinamurthy, P. Seshakumar, B. Prabavathy, and C. Babu. A novel indexing scheme for efficient handling of small files in hadoop distributed file system. In *Computer Communication and Informatics (ICCCI), 2013 International Conference on*, pages 1–8. IEEE, 2013.

[21] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[22] J. Chen, D. Wang, L. Fu, and W. Zhao. An improved small file processing method for hdfs. *International Journal of Digital Content Technology and its Applications*, 6(20):296–304, 2012.

[23] A. Chervenak, J. M. Schopf, L. Pearlman, M.-H. Su, S. Bharathi, L. Cinquini, M. D'Arcy, N. Miller, and D. Bernholdt. Monitoring the earth system grid with mds4. In *e-Science and Grid Computing, 2006. e-Science'06. Second IEEE International Conference on*, pages 69–69. IEEE, 2006.

[24] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*, volume 10, page 20, 2010.

[25] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI\'04*, pages 137–150, 2005.

[26] B. Dong, J. Qiu, Q. Zheng, X. Zhong, J. Li, and Y. Li. A novel approach to improving the efficiency of storing and accessing small files on hadoop: a case study by powerpoint files. In *Services Computing (SCC), 2010 IEEE International Conference on*, pages 65–72. IEEE, 2010.

[27] B. Dong, Q. Zheng, M. Qiao, J. Shu, and J. Yang. Bluesky cloud framework: an e-learning framework embracing cloud computing. In *Cloud Computing*, pages 577–582. Springer, 2009.

[28] B. Dong, Q. Zheng, F. Tian, K.-M. Chao, R. Ma, and R. Anane. An optimized approach for storing and accessing small files on cloud storage. *Journal of Network and Computer Applications*, 35(6):1847–1862, 2012.

[29] D. Q. Duffy, J. L. Schnase, J. H. Thompson, S. M. Freeman, and T. L. Clune. Preliminary evaluation of mapreduce for high-performance climate data analysis. 2012.

[30] W. Fang, V. Sheng, X. Wen, and W. Pan. Meteorological data analysis using mapreduce. *The Scientific World Journal*, 2014, 2014.

[31] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[32] L. Jiang, B. Li, and M. Song. The optimization of hdfs based on small files. In *Broadband Network and Multimedia Technology (IC-BNMT), 2010 3rd IEEE International Conference on*, pages 912–915. IEEE, 2010.

[33] John Gantz, David Reinsel. The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East, Dec 2012.

[34] J. Kepner, W. Arcand, D. Bestor, B. Bergeron, C. Byun, V. Gadepally, M. Hubbell, P. Michaleas, J. Mullen, A. Prout, et al. Achieving 100,000,000 database inserts per second using accumulo and d4m. *arXiv preprint arXiv:1406.4923*, 2014.

[35] N. Laptev, K. Zeng, and C. Zaniolo. Very fast estimation for result and accuracy of big data analytics: The earl system. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 1296–1299. IEEE, 2013.

[36] X. Li, Y. Wang, Y. Jiao, C. Xu, and W. Yu. Coomr: cross-task coordination for efficient data management in mapreduce programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 42. ACM, 2013.

[37] G. Mackey, S. Sehrish, and J. Wang. Improving metadata management for small files in hdfs. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–4. IEEE, 2009.

[38] E. H. Neilsen Jr. The sloan digital sky survey data archive server. *Computing in Science and Engineering*, 10(1):13–17, 2008.

[39] S. M. Sawyer, B. D. O'Gwynn, A. Tran, and T. Yu. Understanding query performance in accumulo. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6. IEEE, 2013.

[40] R. Sen, A. Farris, and P. Guerra. Benchmarking apache accumulo bigdata distributed table store using its continuous test suite. In *Big Data (BigData Congress), 2013 IEEE International Congress on*, pages 334–341. IEEE, 2013.

[41] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1013–1020. ACM, 2010.

[42] F. Wang and M. Liao. A map-reduce based fast speaker recognition. In *Information, Communications and Signal Processing (ICICS) 2013 9th International Conference on*, pages 1–5. IEEE, 2013.

[43] Y. Wang, H. Fu, and W. Yu. Cracking down mapreduce failure amplification through analytics logging and migration. In *Parallel & Distributed Processing (IPDPS), 2015 IEEE 29th International Symposium on*. IEEE, 2015.

[44] Y. Wang, X. Que, W. Yu, D. Goldenberg, and D. Sehgal. Hadoop acceleration through network levitated merge. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 57. ACM, 2011.

[45] Y. Wang, C. Xu, X. Li, and W. Yu. Jvm-bypass for efficient hadoop shuffling. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 569–578. IEEE, 2013.

[46] B. White, T. Yeh, J. Lin, and L. Davis. Web-scale computer vision using mapreduce for multimedia data mining. In *Proceedings of the Tenth International Workshop on Multimedia Data Mining*, page 9. ACM, 2010.

[47] T. White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[48] K. Wiley, A. Connolly, J. Gardner, S. Krughoff, M. Balazinska, B. Howe, Y. Kwon, and Y. Bu. Astronomy in the cloud: using mapreduce for image co-addition. *Astronomy*, 123(901):366–380, 2011.

[49] C. Xu, R. Goldstone, Z. Liu, H. Chen, B. Neitzel, and W. Yu. Exploiting analytics shipping with virtualized mapreduce on hpc backend storage servers.

[50] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[51] Y. Zhang and D. Liu. Improving the efficiency of storing for small files in hdfs. In *Computer Science & Service System (CSSS), 2012 International Conference on*, pages 2239–2242. IEEE, 2012.