Towards High-Performance Big Data Processing Systems

2018

Hong Zhang
University of Central Florida

TOWARDS HIGH-PERFORMANCE BIG DATA PROCESSING SYSTEMS

by

HONG ZHANG
M.S. University of Wyoming, 2015

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2018

Major Professor: Liqiang Wang

ABSTRACT

The amount of generated and stored data has been growing rapidly, It is estimated that 2.5 quintillion bytes of data are generated every day, and 90% of the data in the world today has been created in the last two years. How to solve these big data issues has become a hot topic in both industry and academia.

Due to the complex of big data platform, we stratify it into four layers: storage layer, resource management layer, computing layer, and methodology layer. This dissertation proposes brand-new approaches to address the performance of big data platforms like Hadoop and Spark on these four layers.

We first present an improved HDFS design called SMARTH, which optimizes the storage layer. It utilizes asynchronous multi-pipeline data transfers instead of a single pipeline stop-and-wait mechanism. SMARTH records the actual transfer speed of data blocks and sends this information to the namenode along with periodic heartbeat messages. The namenode sorts datanodes according to their past performance and tracks this information continuously. When a client initiates an upload request, the namenode will send it a list of "high performance" datanodes that it thinks will yield the highest throughput for the client. By choosing higher performance datanodes relative to each client and by taking advantage of the multi-pipeline design, our experiments show that SMARTH significantly improves the performance of data write operations compared to HDFS. Specifically, SMARTH is able to improve the throughput of data transfer by 27-245% in a heterogeneous virtual cluster on Amazon EC2.

Secondly, we propose an optimized Hadoop extension called MRapid, which significantly speeds up the execution of short jobs on the resource management layer. It is completely backward compatible to Hadoop, and imposes negligible overhead. Our experiments on Microsoft Azure public

cloud show that MRapid can improve performance by up to 88% compared to the original Hadoop.

Thirdly, we introduce an efficient 3-level sampling performance model, called Hedgehog, and focus on the relationship between resource and performance. This design is a brand new white-box model for Spark, which is more complex and challenging than Hadoop. In our tool, we employ a Java bytecode manipulation and analysis framework called ASM [1] to reduce the profiling overhead dramatically.

Fourthly, on the computing layer, we optimize the current implementation of SGD in Spark's MLlib by reusing data partition for multiple times within a single iteration to find better candidate weights in a more efficient way. Whether using multiple local iterations within each partition is dynamically decided by the 68-95-99.7 rule. We also design a variant of momentum algorithm to optimize step size in every iteration. This method uses a new adaptive rule that decreases the step size whenever neighboring gradients show differing directions of significance. Experiments show that our adaptive algorithm is more efficient and can be 7 times faster compared to the original MLlib's SGD.

At last, on the application layer, we present a scalable and distributed geographic information system, called Dart, based on Hadoop and HBase. Dart provides a hybrid table schema to store spatial data in HBase so that the Reduce process can be omitted for operations like calculating the mean center and the median center. It employs reasonable pre-splitting and hash techniques to avoid data imbalance and hot region problems. It also supports massive spatial data analysis like K-Nearest Neighbors (KNN) and Geometric Median Distribution. In our experiments, we evaluate the performance of Dart by processing 160 GB Twitter data on an Amazon EC2 cluster. The experimental results show that Dart is very scalable and efficient.

ACKNOWLEDGMENTS

I would first like to express my sincere gratitude to my adviser, Professor Liqiang Wang for his patient guidance, continuous encouragement and significant support during my Ph.D study. He not only inspired me in research, but also gave me many advices on life and future career.

I would like to thank my current and former committee members, Dr. Damla Turgut, Dr. Jun Wang, Dr. Shunpu Zhang, and Dr. Boqing Gong. I am very grateful for their invaluable advice and patience with me over these days. Genuine thanks to each of them for the extraordinary amount of time and knowledge they were willing to provide on my dissertation.

I dedicate this thesis to my family: my parents Linsheng Zhang and Fenglan Zhao, as well as my wife Rui Guo, for their endless love, selfless care and support. Thanks to my lovely daughter Rosie who inspires me to keep going and brings me full of happiness every day.

I would like to thank all the former and current graduate students in Dr. Liqiang Wang's research group. They are Zixia Liu, Siyang Lu, BingBing Rao, Hongyi Ma, He Huang, Ping Guo, Chao Liang, Zhibo Sun, Yifan Ding, Yandong Li, and Ehsan Kazemy.

Last but not least, I would like to thank all my collaborators for discussions I had with them and ideas they gave to me. They are Dr. Hai Huang, Dr. Chen Xu, Weidong Wang, Yong Yang, Lei Chen, and Xiang Wei.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

The proliferation of massive datasets and the surge of interests in big data analytics have popularized a number of novel distributed data processing platforms such as Hadoop and Spark. How to optimize the performance of big data platforms is a big challenge to a system user.

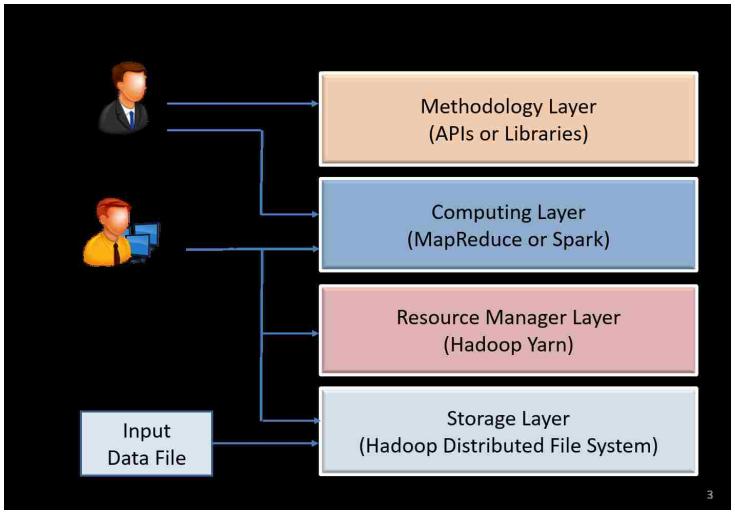


Figure 1.1: Structure of distributed computing platform.

The performance analysis technology for big data platform is the key to explore the hidden value in the big data. The traditional analysis method is to investigate the log information generated by the original big data platform after execution, detect and improve performance bottleneck. But such a method cannot effectively solve complex problems. Therefore, we stratify a big data system into four layers: storage layer, resource management layer, computing layer, and methodology layer, as shown in Figure 1.1 and improve their performance separately. The dissertation is organized based on the optimizations we designed for these four layers.

1.1 Storage Layer

The storage layers employs Hadoop Distributed File System (HDFS), a distributed file system in Hadoop, to store the input and output data for applications. Hadoop provides a distributed, scalable, and portable file system written in Java for the Hadoop framework, called HDFS. There are some major differences from other distributed file systems, *e.g.*, highly fault-tolerant, and can be easily deployed on low-cost hardware. Due to the partitioning of data across commodity hosts, Hadoop can easily distribute computation with tens of thousands of servers in a cluster.

Like other distributed file systems like PVFS, and GFS, HDFS stores input data and file metadata separately as a master/slave architecture. Metadata are stored in a specific server called NameNode; application data are stored across multiple machines called DataNodes. For reliability, these input files are replicated three copies by default.

HDFS contains a single namenode that manages the entire file system, and one or more datanodes serve read and write requests from client systems. HDFS assumes that all nodes in a cluster are homogeneous and can process requests with similar speed. However, in real world, the performance of (*e.g.*, network, disks, and CPU) nodes could be different from one another due to various reasons, *e.g.*, different generations of hardware, different virtual resource allocation, resource contention in virtualized environments. We found that this disparity in performance amongst datanodes within an HDFS cluster can significantly hamper its write performance when handling upload of data files from client local file system, especially when the storage cluster is configured to use replicas.

Therefore, we propose an asynchronous multi-pipeline file write protocol called SMARTH to replace the traditional stop-and-wait protocol in HDFS [2]. Instead of transferring data blocks one by one and waiting for *ACK* (acknowledgement) packets from all datanodes involved in the transmission, SMARTH (Smart HDFS) builds a new pipeline after it finishes sending the current block to

the first datanode in the pipeline so that it can start sending the next data block right away. This new design makes better use of the network capacity of the client as well as the datanodes' accessing bandwidth within the cluster. In order to minimize the time of the file importing process, we introduce a flexible sorting algorithm of datanodes based on real-time and historical datanode accessing status (including network and storage I/O). We employ the heartbeat mechanism to report the data transmission speed on each client to the namenode every three seconds. Based on the collected information, the namenode can give a good estimate of which set of datanodes a client should use for best performance. When the replication factor of an HDFS cluster is greater than one, which is often used in production environments, we optimize the way that a client interacts with each of the datanodes in a pipeline to allow additional parallelism in data transfer. However, this also changes the way that HDFS ensures data fault tolerance, and thus, we revise its fault tolerance method so that the new way is compatible with the asynchronous multi-pipeline protocol.

We simulate various network conditions using bandwidth throttling on Amazon EC2, we demonstrate that the asynchronous multi-pipeline algorithm is able to remove the single pipeline barrier and effectively overlap data transfer in different pipelines for HDFS file write operations. Overall, SMARTH is able to improve the throughput of data transfer by 27- 245% in a heterogeneous virtual cluster of Amazon EC2.

1.2 Resource Management Layer

Resource Management Layer is a resource management platform that allows multiple data processing engines such as interactive SQL, real-time streaming, data science and batch processing to handle data in a single platform. Here we employ Hadoop Yarn for managing computing resources in Hadoop cluster. The fundamental idea of Hadoop Yarn is to separate the resource management and computation process into different daemons. One of the most significant benefits is that we can

execute different computing engines such as MapReduce and Spark in the same platform without interference.

In the YARN architecture, ResourceManager (RM) usually runs on a dedicated machine and supports High Availability (HA) by running one or more RMs in Standby mode. The RM monitors the status and available resources of DataNodes and assign resources among users' applications. The RM also decides how to allocate resources among multi-tenants by different kinds of scheduling algorithms.

However, we found that the original Yarn is inefficient and exists some performance bugs, especially for short job. Although Hadoop is designed to process very large data sets, a majority of jobs are short in the real world. For example, the MapReduce jobs at Google in 2004 took 634 seconds on the average, and over 80% of Yahoo's jobs finished within 10 minutes [3][4][5]. This is mainly due to the input data size being small, especially when it is spread across the entire HDFS cluster and processed in parallel. Moreover, SQL-like query systems, such as Pig and Hive, that operate on top of MapReduce could break a longer running job into a collection of shorter jobs [6][7]. More recently, Uber mode was introduced in Hadoop 2 to specifically deal with jobs with small input size (less than 1 data chunk, to be precise). This special mode runs all tasks of a job within one container. However, even Uber mode is not efficient enough to handle small jobs. We summarize the inefficiencies of running short jobs on Hadoop as follows:

- Hadoop scheduler does not take data locality into account for short job, thus unnecessary data transfer could significantly slow down the execution of short jobs.
- One-time task setup and tear down overheads, which are often negligible in a long running job, can no longer be overlooked for short jobs.
- Piggybacking requests and responses to periodic heartbeat messages is designed for cluster

scalability, but waiting a few seconds here and there adds up quickly. Short-circuiting these paths can be beneficial for short jobs.

- In Uber mode, running all tasks sequentially within a single container does not take full advantage of all local resources.
- Moreover, in Uber mode, intermediate data incur disk I/Os, such as spill operation, could significantly degrade performance.

In this study, we propose an efficient short job optimization on Hadoop, called *MRapid*, to speed up the execution of MapReduce short jobs [8]. In our system, we design two improved modes based on Hadoop: Improved Distributed (D+) mode and Improved Uber (U+) mode. Our contributions are summarized as follows:

- In D+ mode, we design a new scheduler to schedule Map tasks according to the resource distribution situation and data locality. When an ApplicationMaster requests resources from Yarn, instead of waiting for report from NodeManager, our new scheduler allocates resources according to the current resource availability and data distribution in ResourceManager, and responds the request in the same heartbeat, rather than waiting for at least two heartbeats in Hadoop. Our algorithm not only avoids load imbalance problem for short jobs, but also reduces the communication cost. The benefits of spreading out Map tasks and data-locality awareness are significant, especially when a short job can be executed in one wave.
- In U+ mode, rather than executing Map tasks sequentially, we run multiple Map tasks in parallel on the same node. The degree of parallelism depends on the available resources on the node.
- In U+ mode, we cache intermediate data into memory instead of writing them to disk and reading them back later as intermediate data are usually small for short jobs.

- We design a job submission framework, which reserves an ApplicationMaster pool for reuse and avoids the long waiting time to initialize new ones for short jobs.
- For a short job, deciding which mode (D+ or U+) runs faster is a grand research challenge. Our job submission framework handles it by supporting speculative execution. Specifically, the framework can execute an application initially in both D+ and U+ modes. During the execution, a profiler records the execution and data I/O information for each mode. When the framework is confident that one mode is behind the other, the slower one will be terminated. The winner mode can be designated to the short jobs for the future run.

1.3 Computing Layer

Our Computing Layer provides several software frameworks, such as Hadoop MapReduce and Spark, to process big data applications using certain programming models.

Hadoop MapReduce is a computing model for processing large data sets in parallel. MapReduce paradigm is composed of a Map function that performs filtering and sorting of input data and a Reduce function that performs a summary operation. The input data are always split into several blocks, which are executed simultaneously.

Apache Spark[9] is another open source big data processing framework, which claims that when comparing to Hadoop, it runs up to 100 times faster based on in memory processing and 10 times faster on disk [10].

There are several advantages compared to other distributed computing platform like Hadoop and Storm. First of all, along with in-memory data storage, the performance of Spark is several times faster than other big data platforms. Secondly, instead of just “map” and “reduce”, Spark defines a large set of operations (transformations and actions) such as “filter”, “sort”, “groupby”. Thirdly,

it also supports many program languages like Java, Scala and Python. Last but not least, it has an abundant ecosystem to support users to write more complex applications easily.

However, when a user submits an application, there are too many parameters to be considered such as the number of cores per executor, memory size per executor, and the number of executors. How to configure these parameters relies on the user's experience. For instance, considering a case where a cluster contains 10 DataNodes, and each DataNode has 16 cores and 128 GB memory together, we first reserve 1 core and 8 GB memory for OS, Hadoop and Spark. Consequently there are 15 cores and 120 GB per node left to allocate. Then the user can determine how many cores be assigned to each task by "spark.task.cpus", which is 1 by default. We assume 1 core for each task is reasonable for the user's application so that the user can execute 15 tasks in parallel in each DataNode. Therefore we can easily calculate how much memory assigned to each task (120 / 15 = 8 GB), however we also need to consider other overheads, so 7 GB per task makes more sense. If 7 GB is not enough for each task, an "out of memory" error will occur, and the user must increase the memory size per task and recalculate the number of tasks per DataNode. Here we assume 7 GB per task is large enough, then decide how many tasks are executed in each executor. Tasks executed in the same executor can share memory and other resource, but too many tasks being processed in the same jvm could lead to poor performance. Here we assume 5 cores per executor is a reasonable configuration. Since we have 15 cores available in each DataNode, 15 / 5 = 3 executors can be allocated in each node, and each executor contains $5 \times 7 = 35$ GB memory.

we propose an efficient 3-level sampling performance model, called Hedgehog, and focus on the relationship between resource and performance [11]. This design is a brand new white-box model for Spark, which is more complex and challenging than Hadoop. In our tool, we employ a Java bytecode manipulation and analysis framework called ASM [1] to reduce the profiling overhead dramatically.

1.4 Methodology Layer

The methodology layer is to carry out complex operations and design a set of API for a specific domain based on the computing layer. Spark is a powerful open-source processing engine which provides abundant APIs for different domains, such as Spark SQL, Spark Streaming, Spark MLlib, Spark GraphX, and SparkR.

In the context of Spark, the driver of a Spark application can be regarded as a specialized parameter server that updates and distributes parameters in a synchronized way. However, unlike parameter server frameworks that are usually implemented on highly efficient MPI, Spark's *synchronous* iterative communication pattern is based on MapReduce between the driver and workers, which makes it an inefficient platform for machine learning algorithms using SGD.

Therefore, we design an adaptive fast-turn stochastic gradient descent algorithm, called FTSGD for methodology layer, which works more efficiently on the platforms that rely on iterative communication such as Spark and Hadoop to speed up the convergence of machine learning algorithms, *e.g.*, linear regression, logistic regression, and SVM.

Another API we designed in methodology layer is a spatial analyzing system, called Dart, on top of Hadoop and Spark in purpose of solving spatial tasks like K-nearest neighbors (KNN) and geometric median distribution for social media analytics [12].

In big data computing, Hadoop-based systems have advantages in processing social media data[13]. In this study, we use two geographic measures, the mean center and the median center, to summarize the spatial distribution patterns of points, which are popular measurements in geography[14]. The method has been used in a previous study to provide an illustration of social media users' awareness about geographic places[15]. The mean center is calculated by averaging the x and ycoordinates of all points and indicates a social media user's daily activity space. However, it is

sensitive to outliers, which represent a user's occasional travels to distant places. The median center provides a more robust indicator of a user's daily activity space by calculating a point from which the overall distance to all involved points is minimized. Therefore, the median center calculation is far more computing intensive. One social media user's activity space comprises geographic areas in which he/she carries out daily activities such as working or living. The median center thus shows a gravity center of that person's daily life.

The major advantages of Dart lie in: (1) Dart provides a computing and storage platform that is optimized for storing social media data like Twitter data. It employs a hybrid table design in HBase that stores geographic information into a flat-wide table and text data into a tall-narrow table, respectively. Thus, Dart can get rid of the unnecessary reduce stage for some spatial operations like calculating mean and median centers. Such a design not only cuts down users' development expenditures, but also significantly improves computing performance. In addition, Dart avoids load imbalance and hot region problems by using pre-splitting technique and uniform hashes for row keys. (2) Dart can conduct complex spatial operations like the mean center and median center calculations very efficiently. Its methodology layer is a completely flexible and totally extensible module, which provides a better support to the upper analysis layer. (3) Dart provides a platform to help users analyze spatial data efficiently and effectively. Advanced users also can develop their own analysis methods for information exploration.

We evaluate the performance of Dart on Amazon EC2[16] with a cluster of 10 m3.large nodes. We demonstrate that our grid algorithm for the calculation of median center is significantly faster than the algorithm implemented by traditional GIS, and we can gain an improvement of 7 times on a 160 GB Twitter dataset and 9 to 11 times on a synthetic dataset. For instance, it costs 1 minute to compute the mean or the median center for 1 million users.

The rest of this dissertation is organized as follows. In Chapter 2, I briefly review the current

research related to my research work and their limitations. Chapter 3 discusses an innovative asynchronous data transmission approach called "SMARTH" is introduced to greatly improve the write operation's performance in HDFS. We then present an efficient short job optimization on Hadoop, called *MRapid*, to speed up the execution of MapReduce short jobs in Chapter 4. In Chapter 5, we propose an efficient 3-level sampling performance model, called Hedgehog, and focus on the relationship between resource and performance. We design an adaptive fast-turn stochastic gradient descent algorithm, called FTSGD for methodology layer in Chapter 6. In Chapter 7, another API we designed is a spatial analyzing system, called Dart, on top of Hadoop and Spark in purpose of solving spatial tasks like K-nearest neighbors (KNN) and geometric median distribution for social media analytics. Finally this dissertation is concluded in Chapter 8.

CHAPTER 2: LITERATURE REVIEW

In this section, I briefly review the current research related to my research work and their limitations with regards to the performance improvement for big data systems such as Hadoop and Spark.

This dissertation is related to the four research areas in optimizing performance of a big data platform: (1) HDFS System and Data Uploading; (2) Resource Management and Scheduling; (3) Performance Modeling; (4) Structured Data Store.

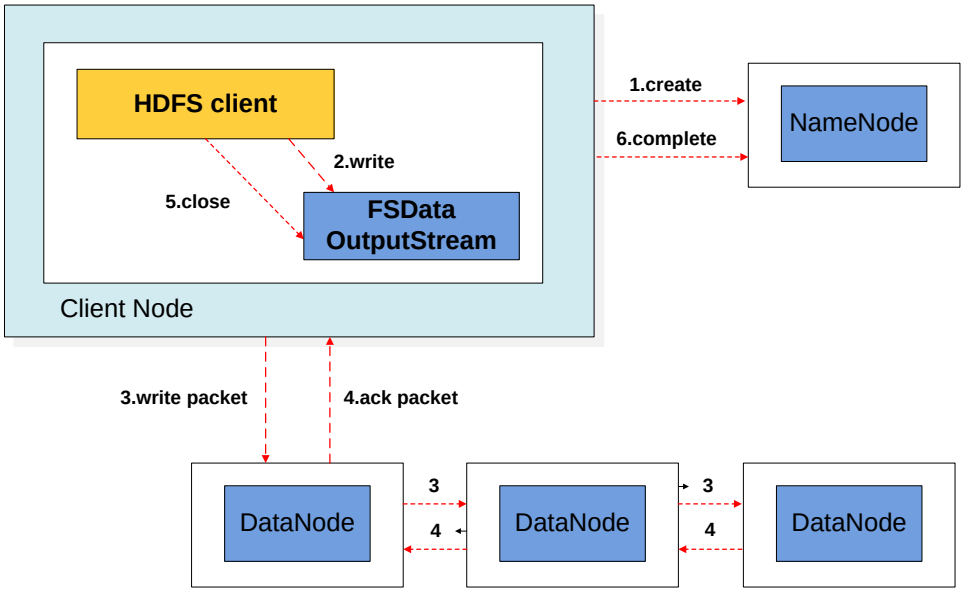


Figure 2.1: Workflow of an HDFS file write operation.

2.1 HDFS System and Data Uploading

An HDFS cluster is comprised of a namenode and one or more datanodes. In this section, we give a comprehensive analysis about how a client communicates with the namenode and datanodes when uploading data to HDFS. As shown in Figure 2.1, there are 6 steps to upload data from a local file system into HDFS.

1. **Creating a file into the file system's namespace.** The client first makes a `create()` HDFS call, which results in a `ClientProtocol` RPC being invoked to create a new file on the namenode. Before the creation of the file in the namespace, the namenode conducts several checks, *e.g.*, whether the file already exists, whether the user has the right to create the file, and whether safe mode is disabled. If all these checks pass, the namenode would create the corresponding file in the file system's namespace; otherwise it would throw an exception.

2. **Splitting data into packets and inserting into a data queue.** To write data to HDFS, client applications consider the data file as a standard output stream. This data stream is fragmented into blocks, each of which has a default size of 64MB. In turn, each block is split into 64KB packets by default when being transmitted onto the network. When the client writes a new block, a `DataStreamer` thread would send an `addBlock()` call to the namenode to ask for a new block ID and the datanode IDs to store the block. After the corresponding packets are generated, the client sends these packets to a FIFO queue and then to the datanodes.

3. **Sending packets to Datanodes.** `DataStreamer` uses the datanode IDs to build a pipeline between the client and these datanodes, streams the packets to the first datanode in the pipeline one by one, and stores these packets into another queue called ACK queue in case some datanodes require retransmitting due to packet loss. When the first datanode receives a packet, it verifies the packet's checksum, stores the packet, and transfers it to the next datan-

ode in the pipeline. This procedure will repeat until the packet reaches the last datanode at the end of the pipeline.

4. **Sending acknowledgement (ACK) back to the client.** When the last datanode obtains the packet, it would send an ACK through the pipeline in a reverse order. The client has a thread called `PacketResponder` that is responsible for receiving response ACKs. If the `PacketResponder` thread receives a packet ACK from all datanodes, it removes this packet from the ACK queue.

5. **Closing the output stream.** When the client has flushed all data into the output stream, it calls `close()` on the stream, and waits for all packets' ACKs.

6. **Completing file write.** When all packets' ACKs are received by the `PacketResponder` thread, it wakes up the client. The client would send a complete signal to the namenode to complete this file write operation.

In Steps 3 and 4, the client has to wait until it received all ACKs through the pipeline, during which the client could not optimally make use of network capacity.

Although a rich set of research has been published on improving the performance of Apache Hadoop nowadays, there is little work in literature to analyze and improve the file transmission paradigm in the HDFS architecture. Xu et al.[17] tries to figure out a cost model to describe the data import and verify this cost model with practical evaluations. In their approach, Instead of opening an input stream to the local file and passing it along to the first datanode through a socket, the original data storage can be directly accessed by datanodes.

There are some literatures related to file write that mainly focus on adjustments to Hadoop parameters and codes to adapt HDFS to a specific scenario. For instance, Shafer et al. [18] analyze the performance of HDFS, and find out bottlenecks existing in the Hadoop implementation that result

in inefficient HDFS usage. Their paper focuses on adjustments of Hadoop parameters to boost the overall efficiency of MapReduce applications. CoHadoop[19] is a lightweight extension of Hadoop that controls where data are stored. It uses hints given by applications to locate data files to improve efficiency. HDFS+[20] is an extended distributed file system from existing HDFS that can accept concurrent writes with multi data sources. In HDFS+, files are divided into fragments not sent in a sequence order, instead, each fragment can be written individually by a client.

A number of other research work have been proposed to make Hadoop more efficient than the original Hadoop. Islam et al.[21] introduce a novel design of HDFS using Remote Direct Memory Access (RDMA) on InfiniBand. The design is able to provide low-latency and high throughput for HDFS write operations as it leverages the RDMA capability of high performance network like InfiniBand. Yee et al.[22] introduce a generic socket API called Hadoop Filesystem Agnostic API (HFAA) to allow Hadoop to integrate with any distributed file system over TCP sockets. This socket API can eliminate the demand to customize Hadoop's Java implementation, and move the implementation responsibilities to the file system. Hadoop-A[23] introduces a novel network-levitated merge algorithm to merge data without repetition and disk access to optimize data processing throughput of Hadoop.

2.2 Resource Management and Scheduling

Yarn, a cluster resource manager, is a key component of Hadoop 2, and MapReduce is one of computing frameworks that runs on Yarn. In this section, we give a comprehensive overview of the job submission process. As shown in Figure 2.2, there are 6 steps to submit a job.

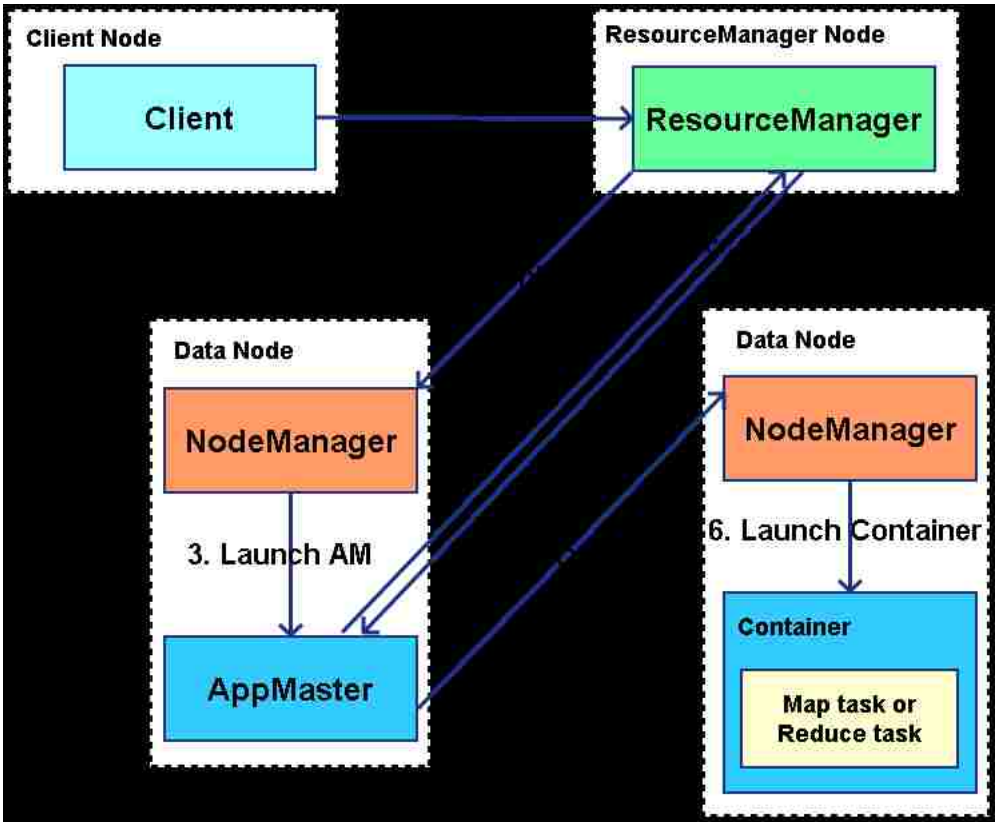


Figure 2.2: Hadoop job submission.

1. *Job Submission:* To submit a new job, a client first communicates with the ResourceManager (RM) to generate a new job ID. which in turn checks the specification of the job, uploads input splits, job Jar file, and configuration to HDFS. The client then submits the job to the RM.

2. *ApplicationMaster (AM) Allocation:* When the RM receives the job submission request, the scheduler allocates a container to set up and launches an AM instance.

3. *Launching AM:* The designated NodeManager (NM) of the allocated container launches AM for the job. Once AM is started, it downloads input splits, job Jar file, and configuration from

HDFS, and initializes itself.

4. *Request Containers:* If the job is not configured to run in Uber mode, the AM requests containers for Map and Reduce tasks from the RM, which schedules resources based on data locality that allocates each task to be near its input data. Hadoop employs CapacityScheduler by default, which allows multiple tenants to share a large cluster and allocate resources under constraints of specified capacities for each user.

5. *Task Assignment:* After tasks have been assigned to run in certain containers by the RM's scheduler, the AM starts the containers by contacting the corresponding NMs.

6. *Task Execution:* After downloading configuration and Jar file from HDFS, the Map or Reduce task is executed as a Java application in JVM.

Submitting job in Hadoop system is inefficient and time-consuming due to creating containers and piggybacking requests and responses to transfer periodic heartbeat messages.

There are four research areas in optimizing performance related to our research: short job scheduling [5, 24, 25], data-locality awareness [26, 27, 28], cache mechanism [29], and reusing resource [30].

Elmeleegy [5] designed a system called Piranha that avoids checkpointing intermediate results to disk. It also supports a simple fault-tolerance mechanism, and employs self-coordination to reduce the cost of high-latency polling protocol. However, this system reduces only cost of spilling data into disk, and the Uber mode is not considered. Yao *et al.* [24] propose a job-size based scheduling algorithm. It leverages the knowledge of workload patterns to reduce average job response time by dynamically tuning the resource sharing among users. But this approach cannot reduce useless overhead caused by Hadoop itself. Yan *et al.* [25] implement an optimized version of Hadoop to reduce the time cost during the initialization and termination stages of a job, and replace the

pull-model task assignment mechanism with a push-model approach. This system just reduces the communication between Driver to NameNode and JobTrack to TaskTrack, but cannot reuse previous jobs' execution environment to speed up.

Hammoud *et al.* [26] designed a Locality-Aware Reduce Task Scheduler (LARTS), which collocate Reduce tasks with the maximum required data after recognizing input data locations and sizes. This method is useful only when the input data are skewed, and the performance improve is not significant. Zhang *et al.* [27] proposed a next-k-node scheduling (NKS) method to reserve nodes for Map tasks to satisfy node locality policy. It is not enough for short job by just considering data locality. Maestro [28] is another scheduling algorithm that schedules map tasks in two waves: first, it fills the empty slots of each data node based on the number of hosted map tasks; second, runtime scheduling takes into account the probability of scheduling a map task depending on the replicas of the task's input data. But for many short jobs, there is only one wave to be executed.

Spark [31][32] is a fast and general engine that can be deployed on Hadoop Yarn. It organizes data into a distributed data structure called resilient distributed dataset (RDD), which can be cached in memory, and be reused across different computations. But we observed that the performance of Spark on Yarn is still slow for short jobs because of the high overhead to launch containers for AMs and executors.

HJ-Hadoop [30] is designed to exploit multicore parallelism at the intra-JVM level, while limiting the number of JVMs created on each node. In our U+ mode, we employ a similar technique, which executes Map tasks of a container in parallel rather than a sequential way.

Besides optimizing MapReduce performance, Hadoop can be improved in many other aspects, such as network [33], HDFS [2], middleware [34], and query optimization [12].

2.3 Performance Modeling

A Spark program includes 5 hierarchical levels: application, job, stage, task, and operation, which is shown in Figure 2.3. Instead of having job as the highest level in Hadoop, the highest level for Spark is application, which is submitted by a client to the resource manager and is launched in an ApplicationMaster container. An application can contain more than one jobs, the number of which depends on the number of actions, *i.e.*, each action is executed by a job.

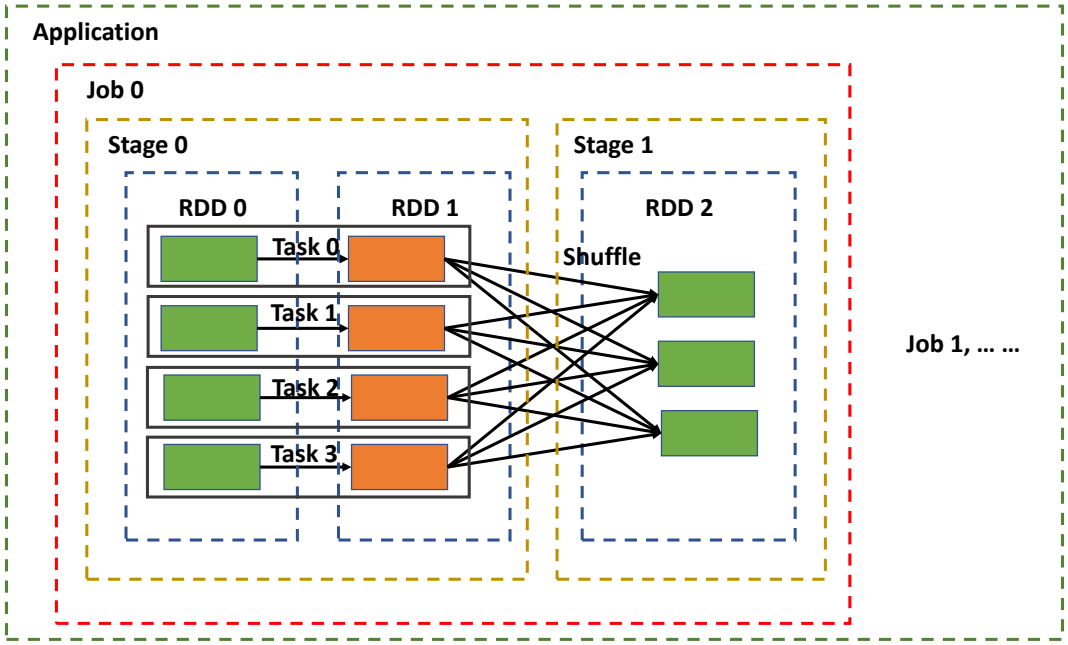


Figure 2.3: 5-Level Structure for Spark Application.

On the application level, all jobs are merged into one package, which makes it easier to design and more efficient to execute a recursive application such as machine learning application. Each job is triggered by one action, which always returns the result to the driver. In each job, due to the number

of shuffling operations, it can be divided into many stages. Spark must shuffle data between two neighboring stages, which is also called wide dependency. A stage may contain multiple tasks, each of which handles one partition of data. Each RDD usually consists of many partitions, thus multiple tasks in each stage may run in parallel on compute nodes. For each task, it still executes several narrow dependent operations, which indicates that operations are executed like pipelining without network shuffling. The lowest level is operation. In general, the lower the level is, the less execution information Spark collects since the overhead is more expensive.

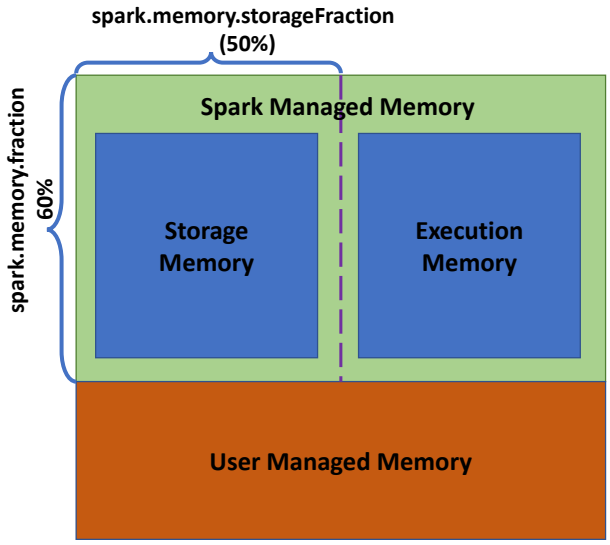


Figure 2.4: Memory Structure in Spark 1.6+.

Ever since Apache Spark version 1.6.0, Spark memory is divided into three regions to manage: user memory, storage memory, and execution memory [35], as shown in Figure 2.4. User memory completely depends on the user-defined function. The quality and feature of the user code has direct effection on this part of memory usage. Storage Memory is used for both cached data and "broadcast" variables. This type of memory is related to data reusing and broadcasting. Execution Memory stores the intermediate shuffling data on both Map side and Reduce side. The boundary

between the storage memory and the execution is not fixed, which means if one kind of memory is insufficient, it can borrow space from the other. How to decide the memory fraction for each memory region is a very challenging problem since the memory usage is totally case by case, and there are so many influencing factors making it hard to analyze.

Starfish [36] is a white-box performance model for Hadoop, which instruments the application and builds the relationship between execution performance and configurations by formulas. However, the overhead of the instrumentation method is too expensive, furthermore, its covering for wide-range applications and formalization makes it inaccurate for some types of applications. There are other existing researches to analyze the performance model but typically not focus on the internal structure and comprehensive process analysis [37, 12, 38, 39, 8].

2.4 Structured Big-data Store

Efficient management of social media data is important in designing data schema. There are two choices when using NoSQL database: tall-narrow, or flat-wide[29]. The tall-narrow paradigm is to design a table with few columns but many rows, while the flat-wide paradigm is to store data in a table with many columns but few rows. Figure 2.5 shows a tall-narrow table and its corresponding realization in the flat-wide format.

For a flat-wide schema, it is easy to extract a single user's entire information in a single row, which can easily fit into a MapReduce program. For a tall-narrow table, each row contains a single record of a user's entire information to avoid data imbalance layout and too much data stored in just a single row.

Key	Q1	Q2	Q3
u1	a	Null	b
u2	Null	c	Null
u3	Null	b	a
u4	b	Null	Null

Key	Qualifier	Value
u1	Q1	a
u1	Q3	b
u2	Q2	c
u3	Q2	b
u3	Q3	a
u4	Q1	b

Figure 2.5: Representations of Horizontal table and Vertical table.

For sparse data, the tall-narrow table is a common way and can support e-commerce type of applications very well[40]. In addition, HBase only splits at row boundaries, which also contributes to the users' choice of tall-narrow tables[29]. Using a flat-wide table, if a single row outgrows the maximum of a region size, HBase cannot split it automatically, which makes data stored in that row overloaded.

In our system, we employ a hybrid schema, in which we store geographic data by a flat-wide table, and store other data like text data by a tall-narrow table. Since numerical location information is significantly smaller than text data, each user's location data can easily fit into a region size (256M by default). Our design can make complex geographic operations more efficient due to removing the reduce stage from a MapReduce job.

Some GIS over Hadoop and HBase have been designed to provide convenient and efficient query processing. However, they do not support complex queries like geometric median. A few systems employ Geographic Index like grid, R tree, and Quad tree to improve the processing, which are,

unfortunately, not helpful for calculating geometric median efficiently.

SpatialHadoop[41] extends Hadoop and consists of four layers: language, storage, MapReduce, and operations. The language layer supports a SQL-like language to simplify spatial data query. The storage layer employs a two-level index to organize data globally and locally. The MapReduce layer allows Hadoop programs to exploit index structure. The operations layer provides a series of spatial operations like range query, KNN, and join. Hadoop-GIS[42] is a spatial data warehousing system that also supports a query engine called REQUE, and utilizes global and local indexes to improve performance. MD-HBase[43] is a scalable data management system based on HBase, and employs a multi-dimensional index structure to sustain an efficient insertion throughput and query processing. However, these systems are incapable of offering a good data organization structure for social network like Twitter, and their index strategies cannot calculate the geometric median efficiently due to an extra load on data management, index creation and maintenance.

CG_Hadoop[44] is a suite of MapReduce algorithms, which covers five different geometry spatial operations, namely, polygon union, skyline, convex hull, farthest pair, and closest pair, and uses the spatial index in SpatialHadoop [41] to achieve good performance. Zhang et al. [45] implements several kinds of spatial queries such as selection, join, and KNN using MapReduce and proves that MapReduce is appropriate for small scale clusters. Lu et al. [46] designs a mapping mechanism that exploits pruning rules to reduce both the shuffling and computational costs for KNN. Liu et al. [47] employs the MapReduce framework to develop a scalable solution to the computation of a local spatial statistic ($G_i^*(d)$). GISQF[48] is another spatial query framework on SpatialHadoop [41] to offer three types of queries, Longitude-Latitude Point queries, Circle-Area queries, and Aggregation queries. Yet, there is no operation or discussion for calculating geometric median on top of Hadoop and HBase.

CHAPTER 3: ENABLING MULTI-PIPELINE DATA TRANSFER IN HDFS

In this work, we introduce an improved HDFS design called SMARTH [1]. It utilizes asynchronous multi-pipeline data transfers instead of a single pipeline stop-and-wait mechanism. SMARTH records the actual transfer speed of data blocks and sends this information to the namenode along with periodic heartbeat messages. The namenode sorts datanodes according to their past performance and tracks this information continuously. When a client initiates an upload request, the namenode will send it a list of "high performance" datanodes that it thinks will yield the highest throughput for the client. By choosing higher performance datanodes relative to each client and by taking advantage of the multi-pipeline design, our experiments show that SMARTH significantly improves the performance of data write operations compared to HDFS. Specifically, SMARTH is able to improve the throughput of data transfer by 27-245% in a heterogeneous virtual cluster on Amazon EC2.

3.1 Asynchronous Multi-pipeline Protocol

In the original HDFS design, when a client wants to write a data block to an HDFS cluster, it receives a list of datanodes from the namenode to form a pipeline. The data block travels from the client to each of the datanodes sequentially, and the client will only mark a block as completed when the ACK packets from all the datanodes in the pipeline are received. Therefore, the effective bandwidth of the pipeline is limited by the slowest datanode in the pipeline.

[1] The content in this chapter was in part reproduced from the following article: Hong Zhang, Hai Huang, Liqiang Wang, SMARTH: Enabling Multi-pipeline Data Transfer in HDFS, 43rd International Conference on Parallel Processing, 2014. The copyright form for this article is included in Appendix A

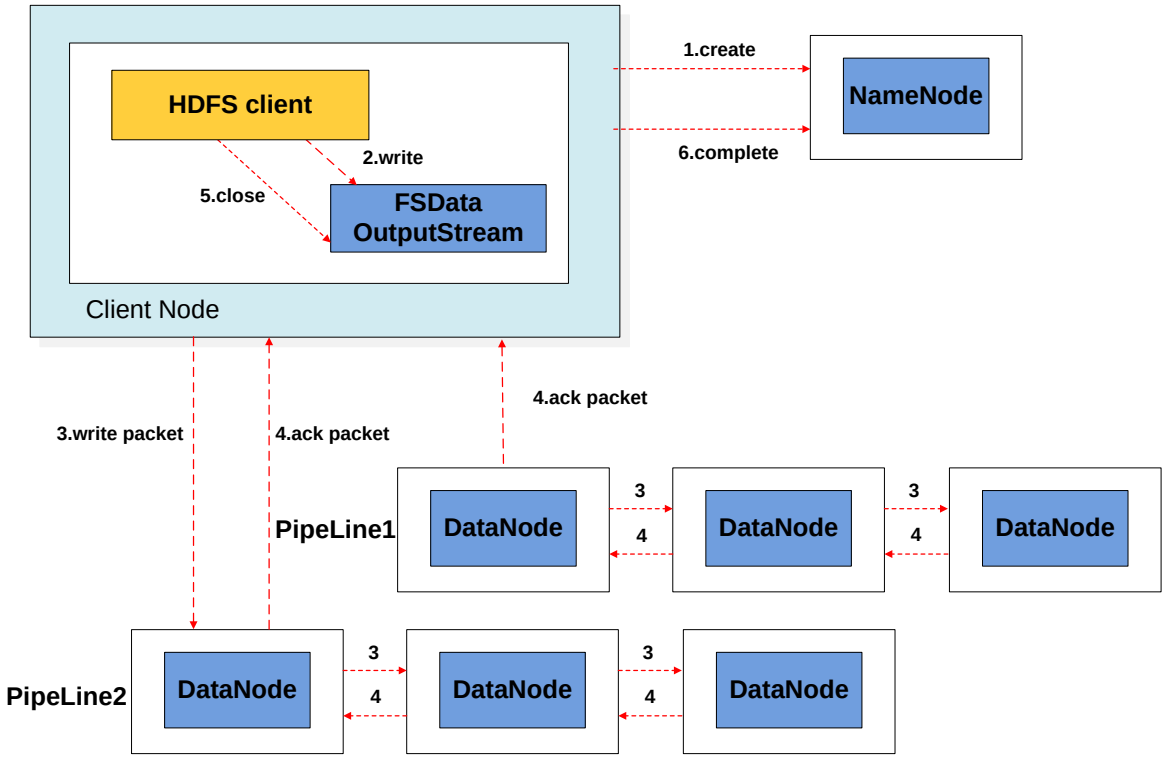


Figure 3.1: Workflow of a SMARTH file write operation.

The way that SMARTH handles data write operations is shown in Figure 3.1. Step 1 is similar to Hadoop. When the client writes a block, it first asks for a block ID and a list of datanodes to store the data. The SMARTH namenode then chooses a high-bandwidth node relative to the client as the first datanode in the pipeline (based on historical information, which we will describe later). In step 2, the client splits data blocks into same size packets and puts them into a data queue.

During data transmission, the client sends these packets to the first datanode, and after storing them locally, this datanode forwards them to the second datanode and so on and so forth until the

last datanode receives all the packets (step 3). When the first datanode receives all the packets of a certain block, it sends back a special ACK packet called `FIRST_NODE_FINISH_ACK (FNFA)` to the client. This packet indicates to the client that the entire block has been received and stored by the first datanode in the pipeline. Instead of waiting for ACKs from the other datanodes, the client continues to send the next data block by requesting another block ID and datanodes from the namenode. This results in a new pipeline being formed for sending the next data block. Additional pipelines can be formed if the client can send packets to the first datanode quicker than the speed that the packets travel to the other datanodes in the pipeline.

After creating a pipeline, we create an ACK queue and a `PacketResponder` thread for it. Each pipeline transfers ACKs back to the corresponding `PacketResponder` thread (step 4). As the `PacketResponder` thread receives an ACK from its pipeline, it removes the corresponding packet from its ACK queue. At the client, we use a set to enumerate all active pipeline objects. When the `PacketResponder` thread receives all ACKs, it will be removed from this set. When the pipeline set is empty, we close output stream (step 5) and complete this uploading (step 6).

Using this method to upload files to HDFS, the client can fully utilize its bandwidth capacity and reduce idle time on waiting for ACK messages. Thus, the speed of the asynchronous pipeline transmission is now determined not by the minimum bandwidth amongst client and datanodes but the network speed between the client and the first datanode in the pipeline. In the following subsections, we describe how SMARTH namenode finds the "best" first datanode for each client while keeping the cluster balanced.

3.2 Global Optimization for Data Transmission

Traditional HDFS represents a network topology as a tree structure [49]. When the client requests a list of datanodes for storing a data block, the namenode chooses the target nodes according to this network topology tree, *e.g.*, to optimize for performance and to maximize data fault tolerance. However, it cannot accurately capture the real-time network condition as this information is usually not directly correlated with the network topology.

In SMARTH, client records the transmission speed of data blocks to all the first datanodes in transfer pipeline that it had communicated before and sends these records to the namenode every three seconds by remote procedure calls (RPCs), following the default heartbeat mechanism in Hadoop. When the client subsequently requests datanodes to place additional data blocks, the namenode utilizes this information to choose a set of best performing datanodes in the cluster according to our global optimization algorithm shown below.

Algorithm 3.1 describes SMARTH namenode's global optimization algorithm for choosing datanodes. When the namenode receives a request to upload files from a client, it calculates the maximum number of pipelines allowed for the client, and assign it to a variable n. Our design selects a datanode randomly from the n best performing nodes for this client as the first datanode so that we can guarantee the bandwidth between the client and the first datanode is relatively higher in the pipeline. The second replica is selected from a different rack and the third replica is placed on the same rack as the second.

Algorithm 3.1 Algorithm for global optimization

```
1: num = the number of active datanodes
2: repli = the number of replica factor
3: n = num / repli // the maximum pipeline size
4: if (namenode has transmission records for the client) then
5:     TopN = top n datamodes in terms of transfer speed
6:     // the number of datanodes we have choosen
7:     results = 0
8:     while (results != repli) do
9:         if (results == 0) then
10:            targets[0] = randomDatanode(TopN)
11:        else if (results == 1) then
12:            targets[1] = randomRemoteRackNode()
13:        else if (results == 2) then
14:            targets[2] = nodeOnSameRack(targets[1])
15:        else
16:            targets[results] = randomDatanode()
17:        end if
18:        results++
19:    end while
20: else
21:    targets = employ the original HDFS method to select datanodes
22: end if
```

3.3 Local Optimization for Data Transmission

Since network status varies all the time, we utilize a local optimization algorithm to sort the datanodes order in pipeline by the newly records and give a chance to test the bandwidth performance of nodes with poor performance previously.

Algorithm 3.2 shows details of local optimization algorithm executes in the client node. We use block transfer records locally to calculate the transmission speed for each datanodes assigned to *TransSpeed*, and employ sort algorithm to reorder the targets set. We calculate a random number *r* between 0 to 1 to decide whether to swap the first datanode with another datanode in pipeline so that we can update the transmission records of that node. In this way, we may keep transmission

Algorithm 3.2 Algorithm for local optimization

```
1: repli = the number of replica factor
2: TransSpeedVector = the transmission speed of every nodes in targets
3: sort targets in descending order by TransSpeedVector
4: r = a random number between 0 to 1
5: if (r > threshold) then
6:     //the target index to switch the first datanode
7:     index = a random integer between 1 to repli − 1
8:     swap(targets[0], targets[index])
9: end if
```

information for all datanodes updated occasionally. In our algorithm, if r is greater than *threshold* that is assigned to 0.8, we use another random integer *index* to choose which datanode to switch with the first one.

3.4 Cost-Benefit Analysis

To pinpoint how SMARTH outperforms HDFS, we analyze a file write operation in details and compare the two designs step by step. Data transfer between a client and datanodes for the original HDFS is shown in Figure 3.2. When the number of replica is greater than one, datanodes will forward each packet to the next datanode along the pipeline until the last datanode receives it. Client will wait for ACKs from all datanodes in the pipeline before it can start sending the next data block.

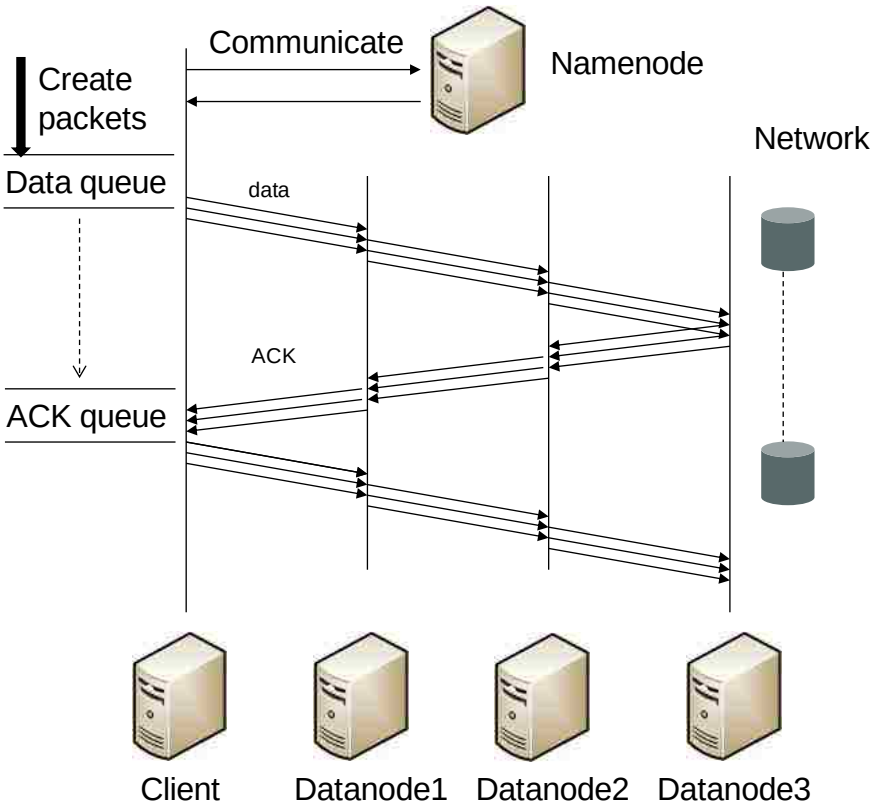


Figure 3.2: Data Transmission of HDFS

Assume that data file size is D, block size is B, and data file size is greater than one block, the file is split into $\lceil D/B \rceil$ blocks. Assume that the packet size is P, the number of packets transferred is $\lceil D/P \rceil$.

Let T_n denote the communication time between client and namenode for each block. Let T_c denote the average production time (read data from local file, compute the checksum and append the data and checksum to a packet) for a packet by the client. When the datanode receives a packet,

it verifies the checksum and writes the packet to the local disk that takes T_w on average. Let B_{min} represent the minimum bandwidth between client and the first datanode and amongst adjacent datanodes. Since the size of ACK packets is smaller than the data packets, and the time of transferring ACKs and the time of sending data packets overlaps, we only need to take the packet transmission time into account.

As the production and the transmission of packets are executed by different threads, there is an overlap between the production time and the transmission time of packets. If the average production time of packets is greater than or equal to the average packet transmission time along the pipeline, there is no packet waiting for sending on data queue. The total production time for all packets is the major factor to the whole importing time. In this scenario, $T_c >= P/B_{min}$, and the total time consuming is shown in Formula (3.1). However, even in the small instance, to produce a packet is very fast compared with the speed to send a packet in our experiments.

$$T = T_n * \ulcorner D/B \urcorner + (T_c + T_w) * \ulcorner D/P \urcorner \tag{3.1}$$

If the packet production time is less than the packet transmission time, there must exist blocking on data queue. So the total cost relies on the minimum bandwidth amongst client and datanodes. In this scenario, $T_c < P/B_{min}$, and Formula (3.2) shows the total time consuming.

$$T = T_n * \ulcorner D/B \urcorner + (P/B_{min} + T_w) * \ulcorner D/P \urcorner \tag{3.2}$$

From the analysis above, we know that the time of importing file is determined by the production time or the transmission time of packets, depending on which is larger.

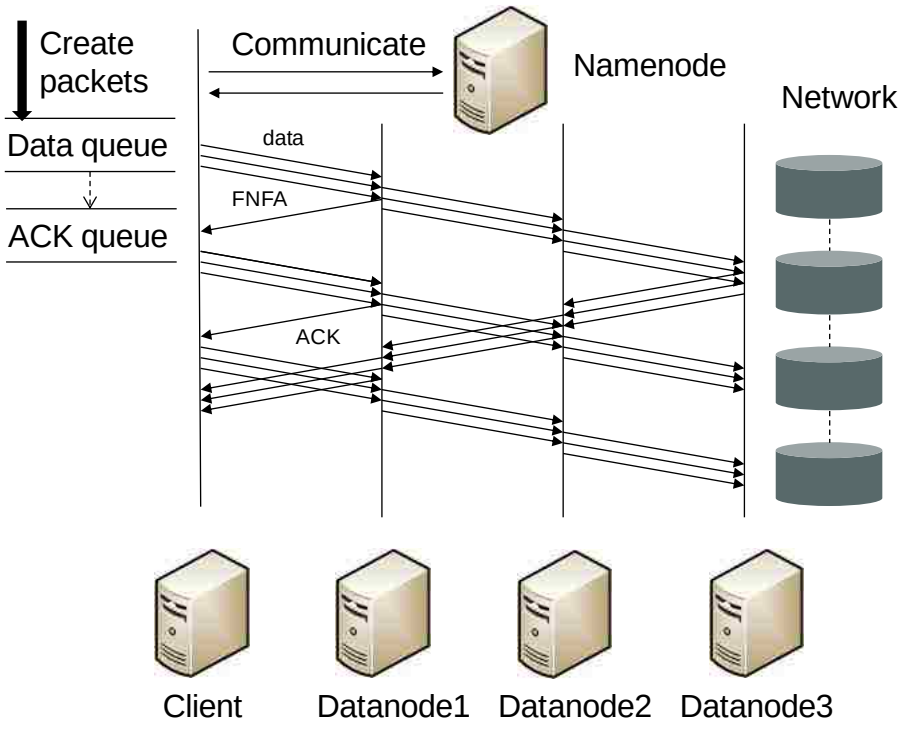


Figure 3.3: Data transmission of SMARTH

Figure 3.3 shows the process of data transmission for SMARTH. When the first datanode receives all packets of a block, it sends a FNFA back to the client, and the client can then create a new pipeline to prepare for transmitting the next block. Assume the bandwidth between the client and the first datanode is B_{max}. If the average production time of packets is greater than or equal to the packet average transfer time from the client to the first datanode ($T_c >= P/B_{max}$), the speed of seeding a packet is slower than the speed of producing a packet. Then there is no blocking in data queue, and the total time consuming is as Formula (3.1) shown.

If the average production time of packets is less than the packet average transfer time from the

client to the first datanode ($T_c < P/B_{max}$), the total cost relies on the bandwidth between the client to the first datanode as Formula (3.3) shows.

$$T = T_n * \ulcorner D/B \urcorner + (P/B_{max} + T_w) * \ulcorner D/P \urcorner \tag{3.3}$$

It is obvious that B_{max} is greater than or equal to B_{min}. So our improved HDFS is more efficient than the existing one. We can also find out that idle time waiting for ACK is reduced when we compare Figure 3.2 with Figure 3.3.

3.5 Fault Tolerance

3.5.1 Fault Tolerance for Original HDFS

Since Hadoop is often deployed on a large cluster of commodity nodes, being able to automatically handle faults is a crucial part of its design. This section provides an overview of the fault tolerance mechanism in original HDFS and then discusses our own fault tolerance approach for the multi-pipeline design in SMARTH.

Algorithm 3.3 shows how a typical process handles errors during uploading files to HDFS. When the client catches an error in the process of transmitting a block, it first checks the validity of parameters, and closes all streams related to the block. Then it moves all packets in ACK queue back to data queue. It picks the primary datanode from active datanodes in pipeline, and uses it to recover the other datanodes. If fails, picks another primary datanode and recover again until recovering the block successfully or throwing an exception. At the end, the client recreates the `ResponseProcessor` thread for receiving remaining ACKs.

Algorithm 3.3 Algorithm for fault tolerance of HDFS

```
1: checks the validity of parameters
2: close all streams related to the block
3: moves all packets in ACK queue back to data queue
4: success = false
5: while (!success) do
6:     if (targets is not empty) then
7:         return an exception
8:     else
9:         primaryNode = the first datanode in targets
10:        add new datanodes to replace error nodes in targets
11:        success=recoverBlock(primaryNode, targets)
12:        if (!success) then
13:            remove primaryNode from targets
14:        end if
15:        recreate block streams
16:    end if
17: end while
18: recreate ResponseProcessor thread
```

3.5.2 Fault Tolerance for Multi-Pipelines

Since we employ an asynchronous multi-pipeline design, we need to replace the original fault tolerance mechanism with a new design.

Algorithm 3.4 Algorithm for fault tolerance of SMARTH

```
1: stop the current block transfer
2: moves all packets in ACK queue back to data queue
3: while (errorPipelineSet is not empty) do
4:     recover one error pipeline as Algorithm 3.3
5:     remove the error pipeline from errorPipelineSet
6: end while
7: start transferring the interrupted block
```

Algorithm 3.4 shows our approach to handle the multi-pipeline fault tolerance. When an error occurs in a pipeline, SMARTH adds the error pipeline into an error pipeline set. It firstly stops the current block sending, and starts a recovery process to recover error pipelines in this set. Each

pipeline's recovery process is similar to the original single pipeline recovery of HDFS. If the error pipeline is recovered, we delete the error pipeline from the error pipeline set. We continue recovering error pipeline until the error pipeline set is empty, then the client restart sending the interrupted block.

3.5.3 Buffer Overflow Problem

In SMARTH, since we employ global optimization and local optimization, the first data node is always a high bandwidth node compared with other datanodes. So the client can send data to the first datanode quickly, but the first datanode cannot send packets quickly to the second datanode. Therefore it is possible that the buffer in the first datanode overflows. When the size of data file is large, and the bandwidth varies considerably from node to node, the buffer of the high bandwidth nodes has higher chance for overflow.

We limit the pipeline size to a maximum number (the cluster size / the number of replica), and if a datanode is already in a pipeline, it cannot be added into other pipelines created by the same client. Then each datanode belongs to only one pipeline, and its buffer is set to be 64 MB, *i.e.*, the default size of block, for each client.

3.6 Experiments

The study was conducted using Amazon EC2's compute instances. Amazon EC2 supports servers of different types such as small, medium, and large instances. These instances differ in the number of cores, the memory allocated to them, bandwidth, and price (see Table 3.1). An Elastic Compute Unit (ECU) is an EC2-specific unit to express the computational performance of a CPU core. 1 ECU is the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or Xeon processor.

Table 3.1: Amazon EC2 instance types

Instance Type	Memory	ECUs	Network
Small	1.7 GB	1	$\approx 216Mbps$
Medium	3.75 GB	2	$\approx 376Mbps$
Large	7.5 GB	4	$\approx 376Mbps$

We use four different clusters in our evaluations. Three of the clusters are homogeneous consisted of one namenode and nine datanodes, i.e., of small, medium, or large instances. The other cluster is heterogeneous consisted of 3 small, 4 medium, and 3 large instance nodes, where one medium instance is the namenode and the others are datanodes. Each node runs CentOS Linux Server 6.2 with kernel 2.6.32-220, and the original Apache Hadoop version 1.0.3. We use Amazon EC2 ephemeral storage to store our data file.

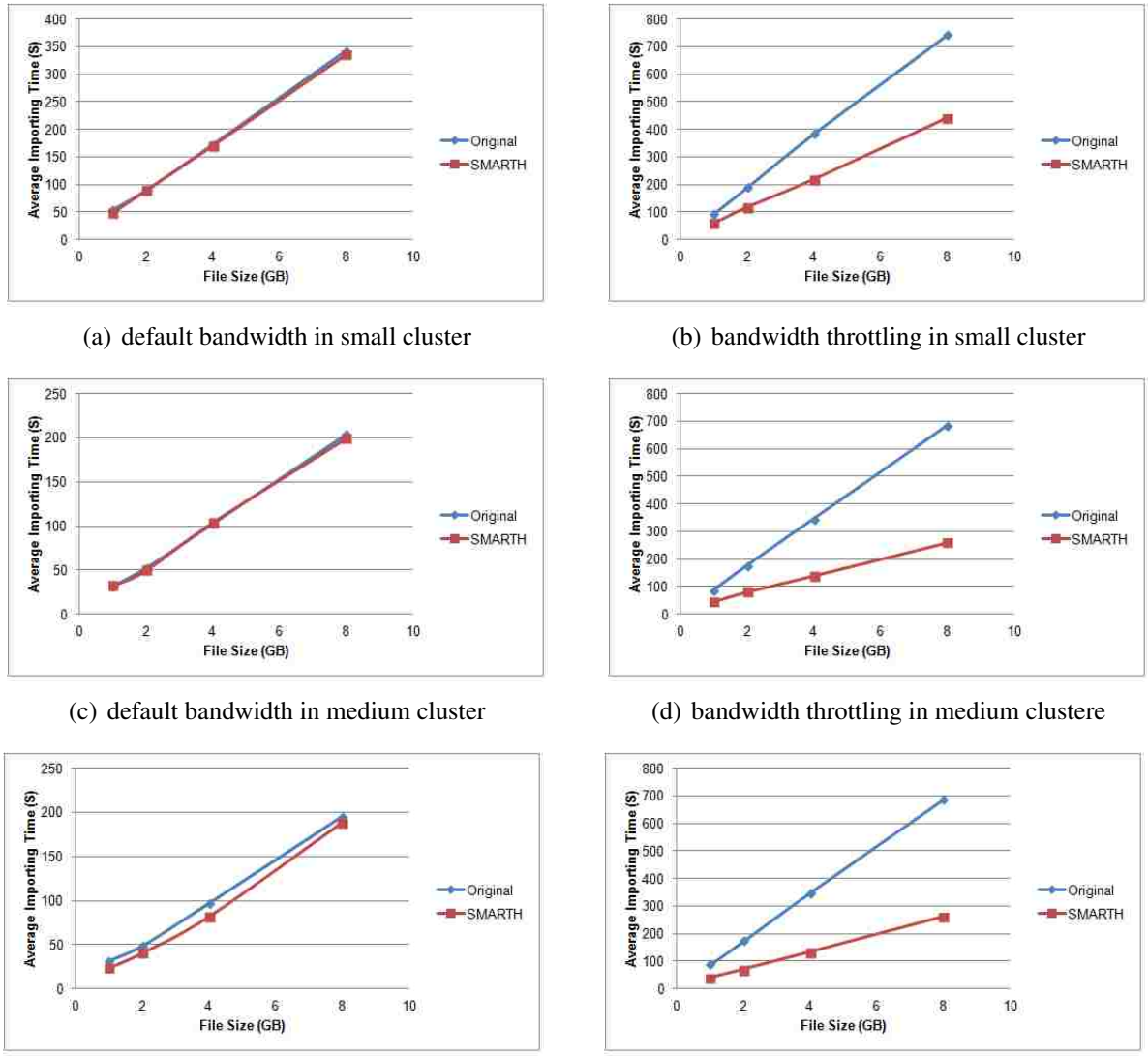
Our goal in this study is to evaluate the impact of various network conditions on both HDFS and SMARTH. We employ a Linux utility called `tc`, which is used to control network traffic, to limit both ingress and egress bandwidth between VMs.

3.6.1 Two-Rack Cluster Scenario

For a large cluster, a common practice is to employ its nodes across multiple racks, or even across multiple data centers, for load balancing and fault tolerance reasons. Network bandwidth between nodes in the same rack is often greater than the bandwidth between nodes across racks. To consistently simulate this behavior (as EC2 does not expose VM's physical location), we throttle the network bandwidth of nodes using `tc`.

The default strategy of HDFS is to place the first replica on the client node itself if the client is a datanode; otherwise, the namenode picks nodes that are not too full or busy. The second replica is

placed on a different rack from the first and the third is placed on the same rack as the second, but on a different node. Although this strategy offers a good reliability, it is at the cost of performance.



(a) default bandwidth in small cluster

(b) bandwidth throttling in small cluster

(c) default bandwidth in medium cluster

(d) bandwidth throttling in medium clustere

(e) default bandwidth in large cluster

(f) bandwidth throttling in large cluster

Figure 3.4: Comparison of uploading time on different clusters with and without network throttling.

In our experiments, our file sizes vary from 1GB to 8 GB, and we measure the time to upload the files in both original HDFS and SMARTH using an HDFS put command. We have performed

these experiments on small, medium, and large instances. Figure 3.4(a) and Figure 3.4(b) show that the file size is proportional to the time consumed when importing file to HDFS and SMARTH in small cluster without and with bandwidth throttling of 100 Mbps between two racks. The same conclusion can be also found in medium and large instances from Figures 3.4(c) and 3.4(d), Figures 3.4(e) and 3.4(f). Due to these results, we only consider the input file size is 8 GB in the rest of the paper when we measure the performance of HDFS and SMARTH.

Figures 3.4(c) and 3.4(e) as well as Figures 3.4(d) and 3.4(f) also show that the file importing performance of large cluster is roughly the same with the performance of medium cluster. That is because the medium cluster and large cluster have the same networking capacity. Figures 3.4(a), 3.4(c), and 3.4(e) show that there is no big gain if the cluster's network status is homogeneous, where network is in the default bandwidth and without throttling.

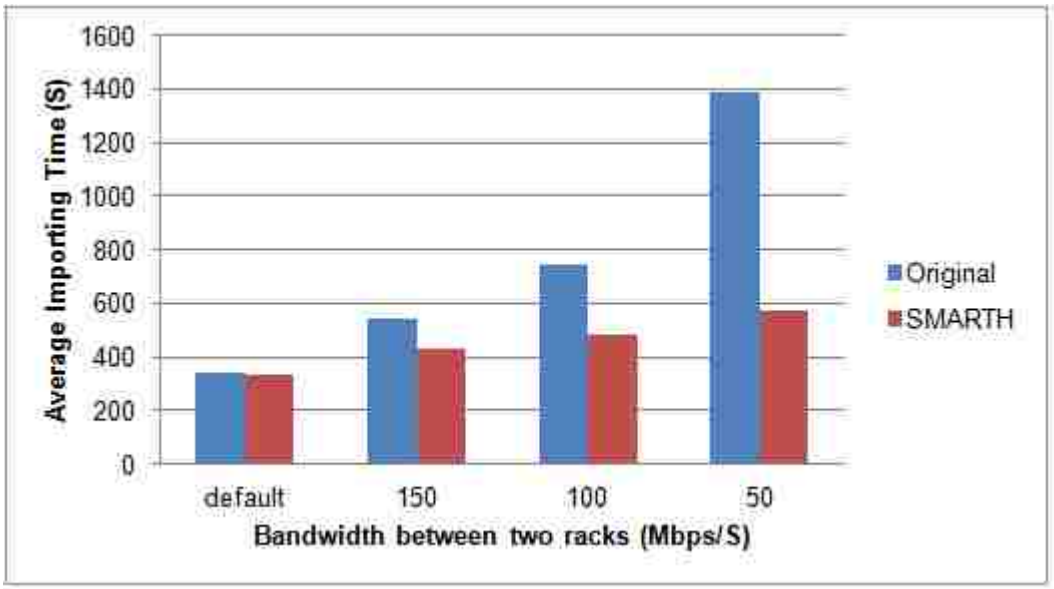


Figure 3.5: Comparison of small instances' uploading time when throttled bandwidth between two racks varies.

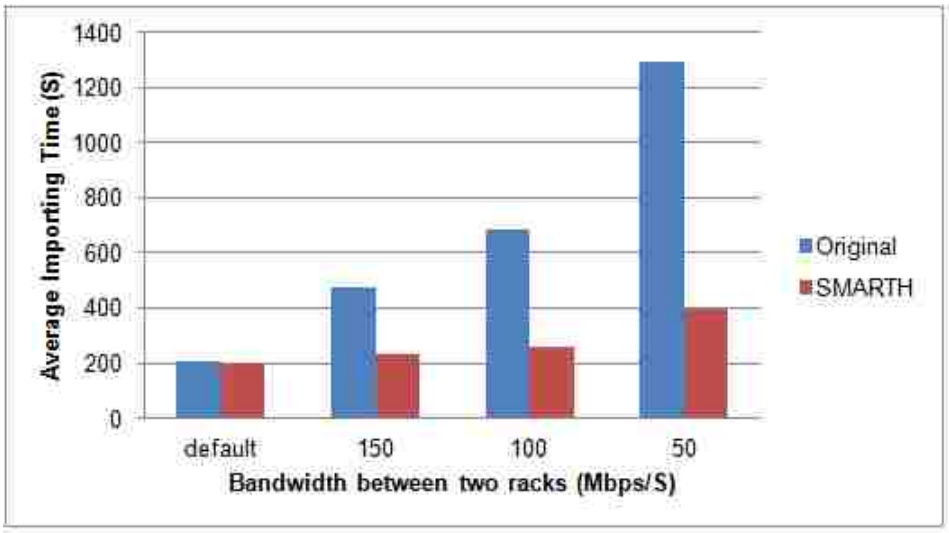


Figure 3.6: Comparison of medium instances' uploading time when throttled bandwidth between two racks varies.

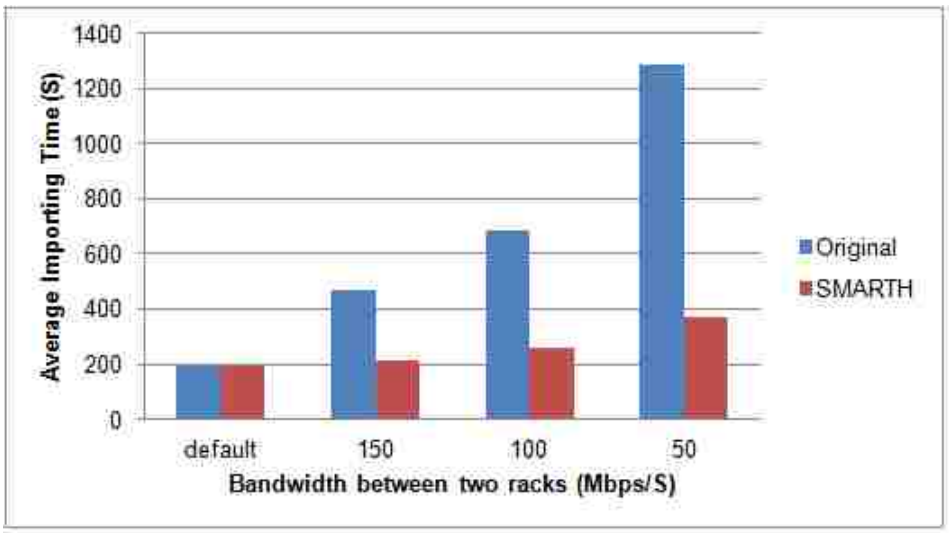


Figure 3.7: Comparison of large instances' uploading time when throttled bandwidth between two racks varies

Figure 3.5 shows the file write times on Hadoop and SMARTH when we throttle the network to different bandwidth in a small cluster. As Figure 3.5 shows, the more we throttle the network, the better the performance of SMARTH is compared to HDFS. The new design of SMARTH gains an improvement of 130% when the bandwidth throttling is at 50 Mbps; even when the bandwidth throttling is 150 Mbps, the performance can improve about 27%. We have measured the file write speed in medium and large clusters and observe similar big gains. Figure 3.6 and Figure 3.7 show that SMARTH achieves an improvement of 225% in medium cluster and outperforms HDFS by 245% in large cluster when the network bandwidth is throttled to 50 Mbps.

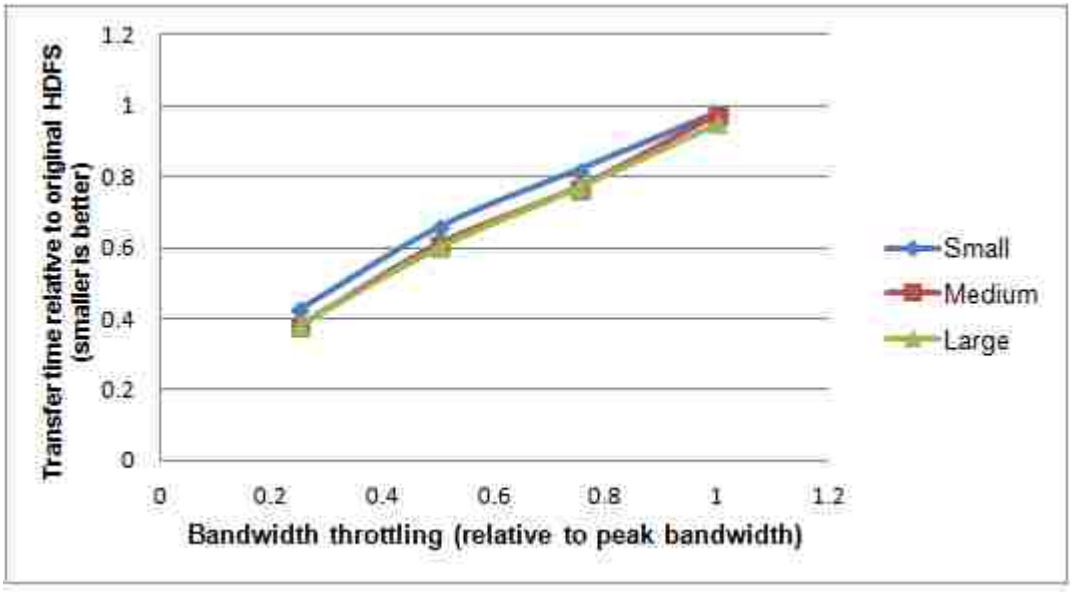


Figure 3.8: Relationship between bandwidth throttling and performance improvement.

Figure 3.8 shows the relationship between how much we throttle the network bandwidth of nodes in small, medium and large clusters and the improvement of SMARTH. The benefit of our design depends on the extent of bandwidth throttling between two racks. When the network bandwidth between nodes in the same rack is much greater than the network bandwidth between nodes in

different ranks, SMARTH can gain more benefit. In a large cluster, where nodes are often allocated in different data centers, network performance between a pair of nodes can vary even more significantly within the cluster.

3.6.2 Bandwidth Contention Scenario

In the real world, the bandwidth between nodes in the same rack still varies all the time, and some other procedures also can occupy the bandwidth and contend with Hadoop program. In this scenario, if some nodes with lower network capacity are selected as datanodes to transfer blocks, they can degrade the performance of file write. In SMARTH, we would select the faster nodes as the first datanode and when the first datanode receives the full block, the client builds a new pipeline to continue the file write in order to avoid the idle wait time of the client network and make the best use of the bandwidth between the client and datanodes.

Figure 3.9 shows the time spent during file write when we vary the number of nodes with 50 Mbps throttling from 0 to 5. As shown in Figure 3.9, even there is only one node whose bandwidth is lower than other datanodes, SMARTH can outperform the traditional Hadoop cluster by 78%. We also can find that the more nodes with lower bandwidth, the more improvement can be gained by SMARTH.

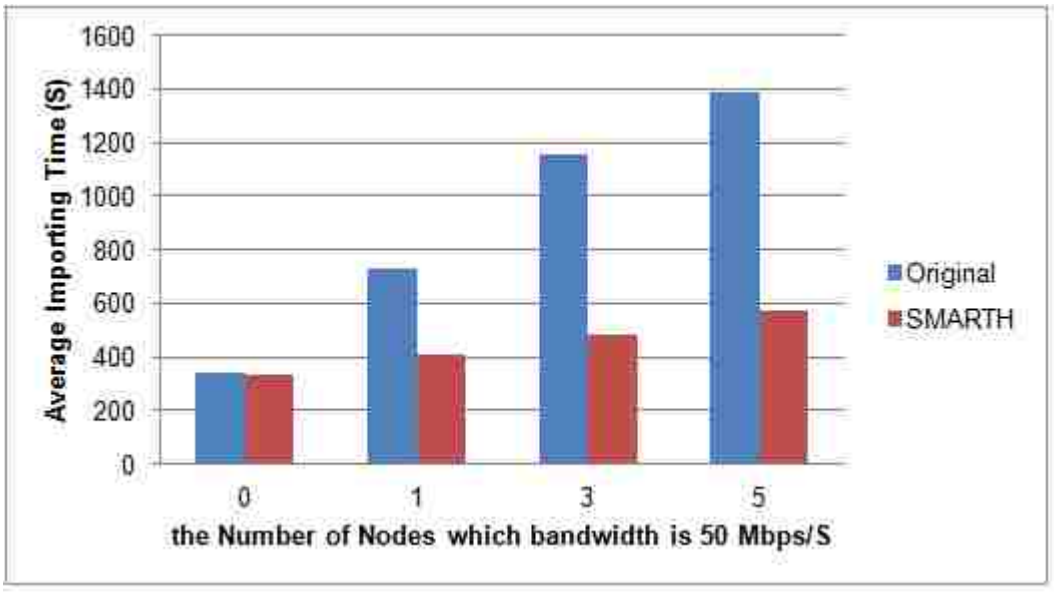
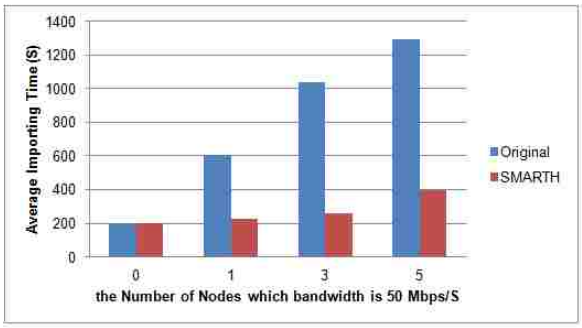
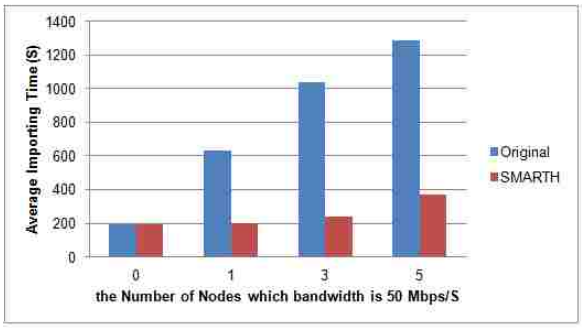


Figure 3.9: Comparison of small instances' uploading time when the number of nodes with 50Mbps throttling varies.



(a) medium cluster

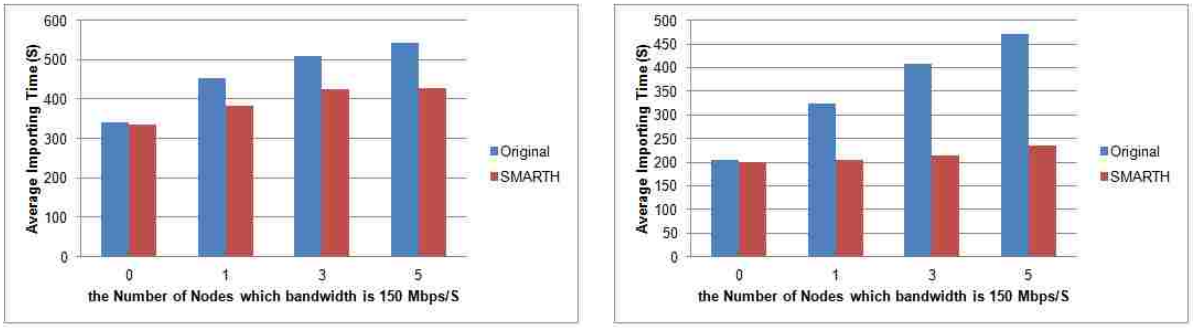


(b) large cluster

Figure 3.10: Comparison of uploading time for medium and large clusters when the number of nodes with 50Mbps throttling varies.

As we would expect, performance gain increases when we evaluate in medium and large clusters due to the big gap between the default bandwidth and the throttling bandwidth. From Figure

3.10(a), we observe an improvement of 167% when uploading a 8 GB data file in medium cluster, we can find the similar result in large cluster from Figure 3.10(b) when only one node's bandwidth is limited to 50 Mbps. The results also illustrate that the medium cluster and large cluster have the similar performance when the bandwidth limitation is the same.



(a) small cluster

(b) medium cluster

Figure 3.11: Comparison of uploading time for small and medium clusters when the number of nodes with 150Mbps throttling varies.

We also test the import time when we vary the number of nodes with bandwidth throttling of 150 Mbps in small and large clusters. From graphs in Figure 3.11(a) and 3.11(b) , the benefit of SMARTH is reduced to 19% in small cluster and 59% in medium cluster compared with the bandwidth throttling of 50 Mbps.

3.6.3 Heterogeneous Cluster Scenario

For power, cost, and pricing reasons, clusters are evolving towards heterogeneous hardware. Heterogeneity also arises due to phased hardware upgrades over years. For example, data center expansion or upgrade will often result in multiple generations of hardware so that network topology may vary, with some routers having lower latency or supporting higher bandwidth than others [50].

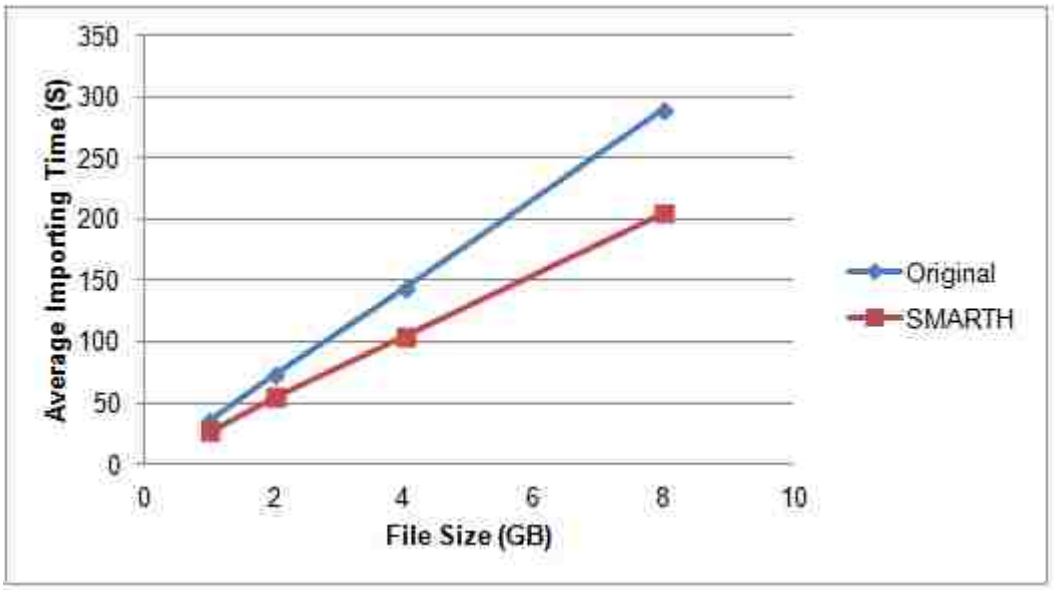


Figure 3.12: Comparison of uploading time of different data size in a heterogeneous cluster.

We repeat the same set of experiments in a heterogeneous cluster consisted of a mixture of small, medium, and large EC2 instances. Without any network throttling, Figure 3.12 shows that it takes 289 seconds to upload an 8 GB file in HDFS, but SMARTH only takes 205 seconds, which is 41% faster.

3.7 Conclusions

Motivated by the increasing popularity of Hadoop applications, in this paper, we introduce an asynchronous multi-pipeline file transfer protocol with a revised fault tolerance mechanism instead of the HDFS's default stop-and-wait single-pipeline protocol. We employ global and local optimization techniques to sort datanodes in pipelines based on the historical data transfer speed. We conduct a series of experiments on Amazon's EC2 by varying the instance type, number of

instances, and network bandwidth. Our experiments reveal significant improvement by 27-245% compared with HDFS.

CHAPTER 4: AN EFFICIENT SHORT JOB OPTIMIZER ON BIG-DATA PLATFORM

In this work, we propose an optimized Hadoop extension called MRapid [1], which significantly speeds up the execution of short jobs. It is completely backward compatible to Hadoop, and imposes negligible overhead. Our experiments on Microsoft Azure public cloud show that MRapid can improve performance by up to 88% compared to the original Hadoop.

Although there is no exact definition, a short job roughly means that its completion time ranges from seconds to minutes, rather than hours. Hadoop's Uber mode gives a more quantitative definition: a small job has less than 10 mappers, only 1 reducer, and the input size is less than the size of one HDFS block. However, this definition still cannot help users decide whether to run MapReduce jobs in Uber mode or not. We consider that the definition of short job is relevant to resource available for users, and the threshold between short job and large job varies depending upon the available resource in the cluster. For instance, if the cluster contains 10 DataNodes, each of which can launch 2 containers, we can run 20 Map tasks in parallel. But if the cluster consists of 100 DataNodes with the same configuration, 200 Map tasks can be executed in one wave.

For a short job running in Hadoop, it is difficult to decide which way is more efficient: distributing all Map tasks to the cluster uniformly or executing them in a single container. Spreading Map tasks to the whole cluster maximizes resource utilization; however, requesting and launching containers will consume a large amount of unnecessary time, and shuffling intermediate data from the Map phase to the Reduce phase is also expensive. An alternative way is to execute all Map and Reduce tasks in a single container in Uber mode, but the current Uber mode executes all tasks sequentially,

[1]The content in this chapter was in part reproduced from the following article: Hong Zhang, Hai Huang, Liqiang Wang, MRapid: An Efficient Short Job Optimizer on Hadoop, IEEE International International Parallel and Distributed Processing Symposium, 2017. The copyright form for this article is included in Appendix B

which is extraordinarily inefficient. Therefore, we design two computing modes: one is a new resource and data-locality aware strategy that distributes and executes Map tasks in parallel in the cluster, which is called the Improved Distributed mode (D+ mode); another is the Improved Uber Mode (U+ mode) that executes Map tasks in a single container in parallel using multiple threads, and stores intermediate data in memory rather than to disk to speed up job execution.

No matter what mode to choose, it is inevitable to launch an AM for each Hadoop job. From experiments, we notice that the time on initializing a short job and launching its AM is expensive compared to the overall execution time of the short job. Therefore, to reuse AM, we introduce a new framework to reserve AM objects in a pool rather than allocating a new one for each short job.

4.1 Distributed Mode

In the D+ mode, our resource and locality aware scheduler allocates Map tasks to different nodes as distributed as possible in order to avoid resource contention like CPU, memory, and disk I/O. Due to data-locality awareness, it also increases the number of data-local Map tasks and reduces data transferring between DataNodes.

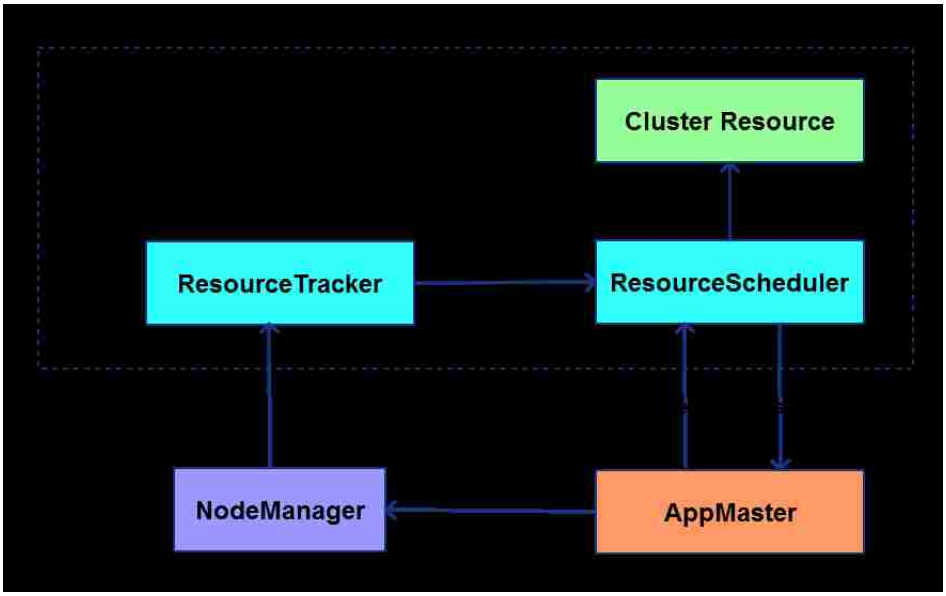


Figure 4.1: Resource request in Hadoop.

Figure 4.1 shows steps how to request containers from RM in the original Hadoop. The AM requests containers and obtains resources from the RM. This request is wrapped into a heartbeat and invoked periodically. The heartbeat contains description of new request, list of released containers, and update information of blacklist nodes.

When the RM receives such a kind of heartbeat, it sends a CONTAINER_STATUS_UPDATE event to the ResourceScheduler (RS). The RS puts the container request to the corresponding queue.

As one of NMs reports its status to the RM by heartbeat, the RM sends a NODE_STATUS_UPDATE event to RS, then the RS allocates available resources of this node to the container request in front of the request queue. Note that one request may ask for multiple containers. The corresponding AM will obtain these containers at the next heartbeat. Then the AM tells the selected NM to start Map or Reduce Tasks. The NM will register launched containers to the RM later.

From the description above, we know that the RM does not respond to the container request immediately, it has to wait until one NM with available resources reports its status, then allocates these resources to the container request. However, such a resource scheduling scheme has several major defects, especially for short jobs. First, waiting for NM status report is a waste of time, which causes the AM cannot obtain resources at the current heartbeat, even there are massive idle DataNodes. The time consumption of communication between the AM and the RM is expensive for short jobs and could not be ignored. Secondly, this scheme can lead to container allocation imbalance, so that some DataNodes may be squeezed with many containers, but others could be idle. Last but not least, this algorithm is not aware of data locality for short jobs. Although this method in the original Hadoop is not bad if the input data are large and spread uniformly in the cluster, lack of data locality is a fatal problem for short job since transferring input data is inevitable especially when the size of cluster is not small.

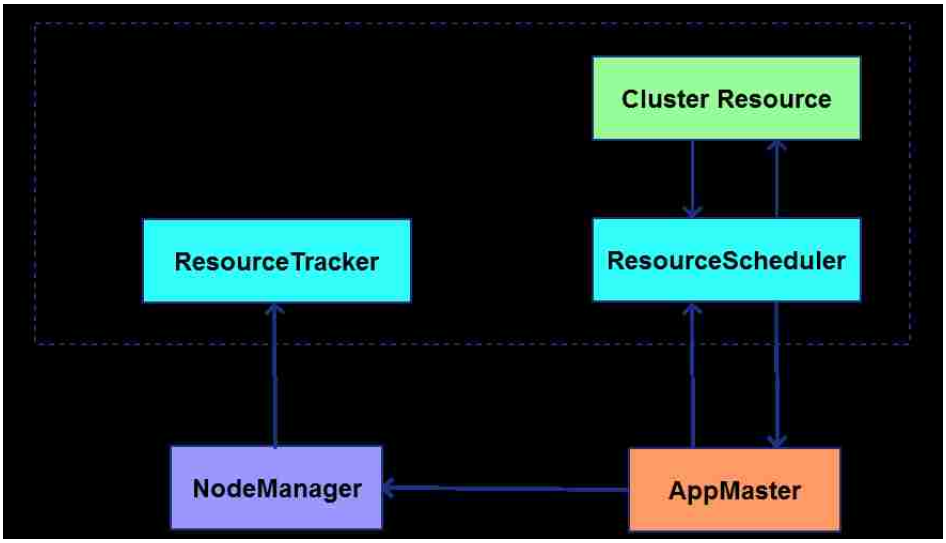


Figure 4.2: Resource request in D+ Mode of MRapid.

Figure 4.2 illustrates our improved distributed mode to allocate resources to small jobs more evenly

and efficiently. The AM sends a request of resources to the RM, which in turn generates a CONTAINER_ STATUS_UPDATE event and sends it to the RS. In Step 2, when the RS realizes that this is a request for short job, instead of waiting for available DataNodes to report their status, the RS can allocate resources from Cluster Resource, which is a special structure designed to store the current resource information of each node and decide how to allocate resources using Algorithm 4.1. The resource status for each node is updated by each heartbeat, so it is sufficient to represent the latest resource status. Step 3 in Figure 4.2 shows that the RS updates the resource usage for every DataNode. After the RS finishes resource allocation, the RM sends resource allocation information immediately to the AM, as shown in Step 4. The rest steps are the same as the original Hadoop to launch containers on the selected NMs.

Algorithm 4.1 Scheduler algorithm for distributed mode

Input: *request*, *nodes*

Output: *response*

```
1: types = {NodeLocal, RackLocal, ANY}
2: for each type in types do
3:     Decide which resource is the current dominant resource;
4:     Sort nodes by available dominant resource in descending order;
5:     for each node in nodes do
6:         for each task in request do
7:             container = getResource(task, node, type);
8:             if (container is not null) then
9:                 response.add(container);
10:                request.del(task);
11:            end if
12:            if (request is empty) then return response
13:            end if
14:        end for
15:    end for
16: end forreturn response
```

Algorithm 4.1 shows our improved CapacityScheduler. The original Hadoop scheduler allocates containers from each available DataNode by a greedy algorithm, which deploys tasks to DataNodes as few as possible. Thus it does not consider data locality and container allocation balance in a global view. In our algorithm, we sort nodes by available dominant resource in descending order so that we assign Map tasks to relatively idle nodes. Dominant resource is a kind of resource such as CPU or memory that has the highest usage ratio in the cluster. Note that our dominant resource

definition is based on the whole cluster, which is different from dominant resource [51] for each user.

HDFS's default replica is three. Its placement policy usually stores one replica on a node in the local rack, another replica on a node in a different rack, and the last on a different node in the same remote rack. Then there are three resource types corresponding to the preferred node of resource request. NodeLocal means that the preferred node is the same with the resource node. RackLocal is the type that the requested node and the resource node are in the same rack. ANY type is that we can designate any resource node to execute Map tasks. So in our approach, we schedule Map tasks to the NodeLocal resource first, then RackLocal, at last ANY in order to take data locality into account adequately until this resource request is satisfied. For each task, we assign resource by "getResource" if the task preferred type (NodeLocal, RackLocal, or ANY) matches the current node with available resources. After one type of resource request has been served, we calculate the dominant resource and sort nodes again to place the current relatively idle nodes in front before serving the next kind of request.

Our D+ mode spreads tasks of a short job across the cluster uniformly, which avoids work overload on specific nodes. In addition, our approach responds to AM requests in one heartbeat, whereas Hadoop usually needs two or more heartbeats. Another important advantage is that our design fully considers data locality before assignment rather than afterwards redistribution, which involves lots of data movement.

4.2 Improved Uber Mode

An Uber task is that the AM uses its own JVM to run the whole Map and Reduce tasks for a short job. Rather than executing each mapper and reducer task in a separated container, the AM

container runs Map and Reduce tasks within its own process to avoid the overhead of requesting, launching, and communicating with remote containers.

Figure 4.3 describes the procedure to run a Hadoop job in the original Uber mode. Since there is only one container available, the AM has to execute Map and Reduce tasks sequentially. Another reason causing inefficiency of the original Uber mode is that intermediate data of Map tasks are spilled to local disks.

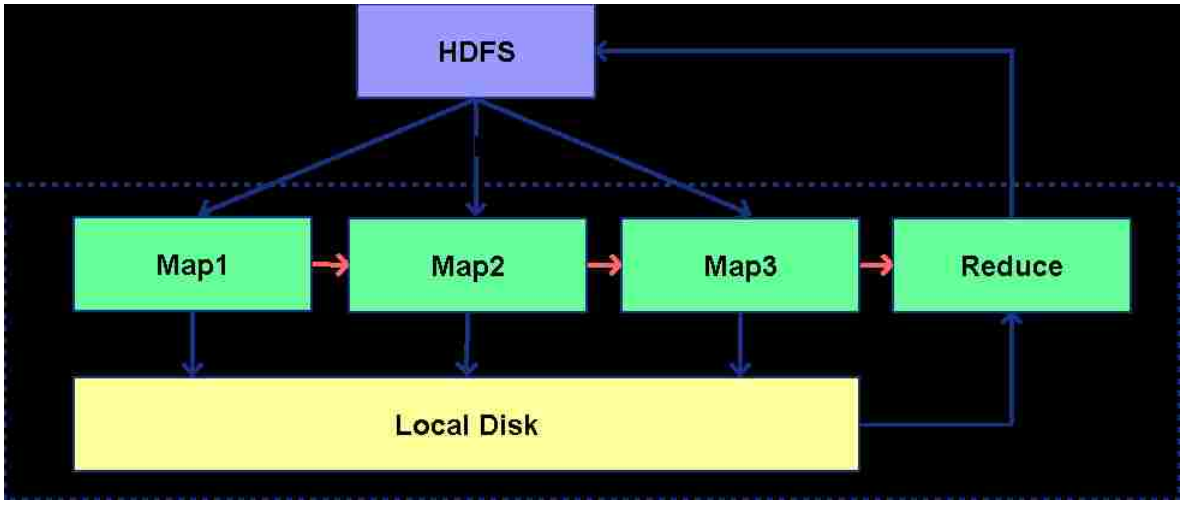


Figure 4.3: Hadoop's Uber mode.

To eliminate these inefficiency problems, we design an improved Uber Mode (U+ mode) that inherits the single container feature from the original Uber Mode, but is extended with the support of multithreading. Figure 4.4 shows the details of U+ mode. When an AM is launched, it parses the job configuration, fetches input data from HDFS, and then executes Map tasks concurrently using multithreading. The number of Maps per wave for the U+ mode (n_u^m) depends on cpu_vcores (n^c, the virtual CPU cores, which can be configured by users) of the AM. Let n_c^m denote the number of Map tasks running simultaneously on each cpu_vcore. Thus, $n_u^m = n^c * n_c^m$ indicates how many

Maps per wave. For a small amount of intermediate data, we store them into memory instead of writing them to local disks. Thus, the Reduce task can fetch results of Map tasks from memory directly to decrease shuffle overhead.

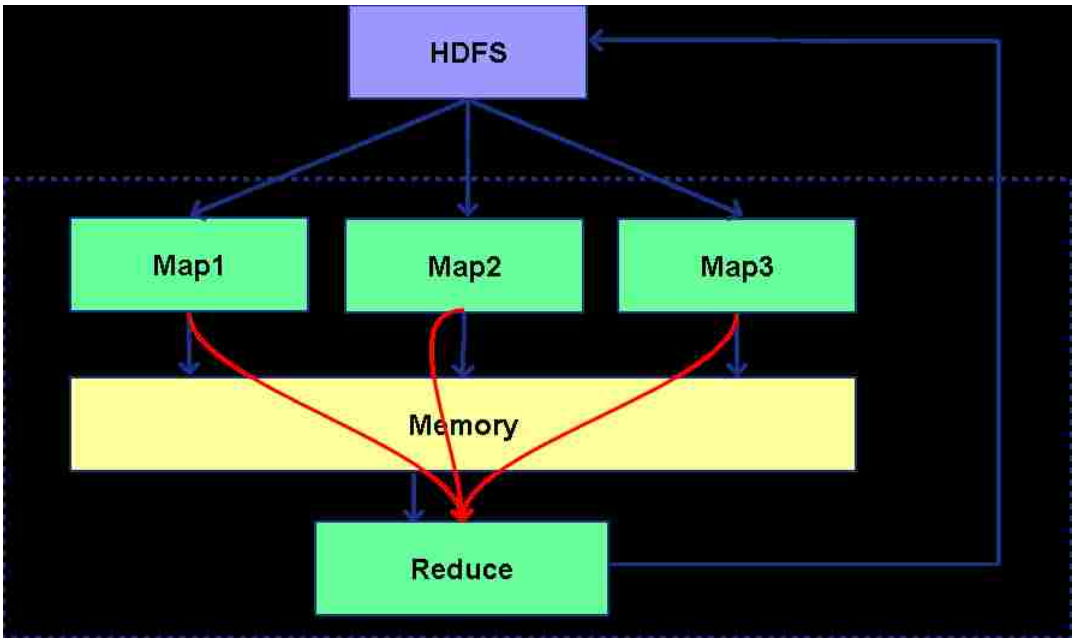


Figure 4.4: U+ Mode in MRapid.

4.3 Job Submitting Framework and Speculative Execution

As shown in Figure 2.2, after a client uploads job files (*e.g.*, jar file, configuration file) to HDFS, it submits the job to the RM. Then the RM launches an AM in a DataNode to manage job execution. The cost to request a container and launch AM is high for a short job, so we design a novel job submission framework using Spring Hadoop [52]. Our framework consists of three major modules. (1) The proxy is used to maintain an AM pool that contains a reasonable number of AMs reserved for short jobs and allocate an AM for each short job. (2) The client module is responsible for

uploading the jar file and configuration files to HDFS and submitting short job to the proxy. (3) The AMSlave module is the module to accept and execute AM from the proxy instead of the RM of the original Hadoop. We implemented a RPC (remote procedure call) to allow the proxy to communicate with the AMs.

Due to the unpredictability of execution time for different kinds of short jobs, we employ a speculative execution mechanism to execute short jobs in both D+ mode and U+ mode. Figure 4.5 shows the workflow of speculative execution in our system.

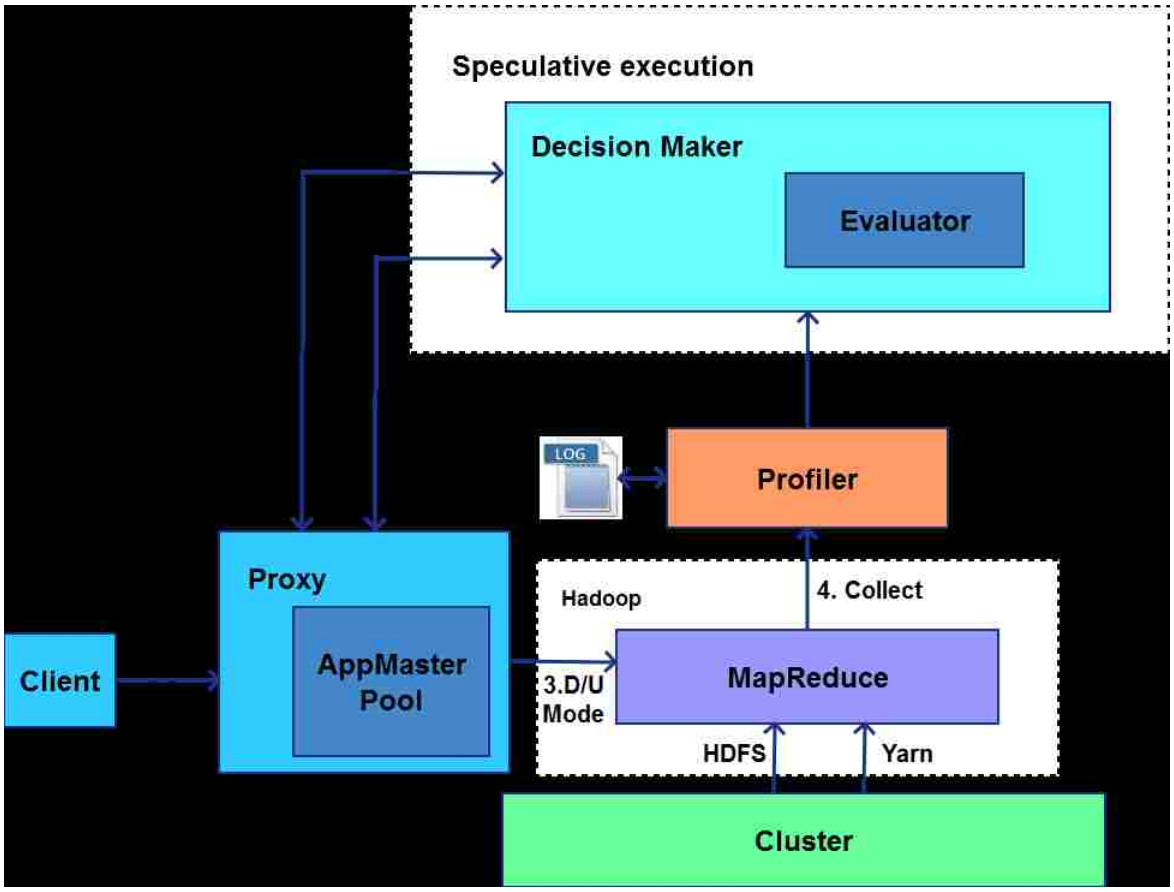


Figure 4.5: Speculative execution in MRapid.

1. **Job Submission:** When Hadoop starts, it launches a proxy service and creates an AM pool that reserves a certain number of AM containers for short jobs. The number of AMs is configured by Hadoop administrator, which is 3 by default. Users can use the client of our submitting framework to submit a short job to the proxy, request a job ID from HDFS, and upload the jar file and configuration files to HDFS.

2. **Pre-decision Making:** When the proxy receives job submission request, it consults the decision maker for which mode will be more efficient based on the execution records of the same job, even if they were executed with different input data.

3. **Launching AM:** If the decision-maker gives a clear answer on which mode is preferred, the proxy chooses one AM container from the pool to submit the job. Otherwise, it submits the job in both U+ and D+ modes.

4. **Profiling:** The designated AM receives the job information from the proxy, then downloads input splits, job Jar file, and configuration from HDFS, and executes the job. We designed a specific Hadoop profiler using ASM [53], which is a small and fast Java bytecode manipulation framework. Our profiler collects Hadoop application execution information including input/output data size and the average execution time for Map and Reduce tasks, and uploads them to HDFS.

5. **Evaluation:** We estimate the total execution time for a job in both U+ mode and D+ mode using the record collected by the profiler. The detailed estimation algorithm will be discussed later. The decision maker evaluates the performance of the two modes, and asks the proxy to terminate the inefficient one.

6. **Terminating Slower Mode:** After the proxy receives a notice from the decision maker, it kills the slower mode and releases allocated resources.

Table 4.1: Notations used in the estimation algorithm

t^{job}	the total execution time for job
t^{AM}	the AM setup time
t^{Map}	the Map phase execution time
$t^{Shuffle}$	execution time of shuffling phase
t^{Reduce}	the Reduce phase execution time
n^m	number of Map tasks
n^c	number of available containers
n^w	number of waves
n_u^m	number of Maps per wave for the U+ mode
t^l	execution time for launching container
t^m	execution time for map sub-phase
d^i	disk input rate
d^o	disk output rate
b^i	bandwidth
s^i	average input data size of Map tasks
s^o	average output data size of Map tasks
t_u	execution time for job in U+ mode
t_d	execution time for job in D+ mode

$$\begin{aligned} t^{job} &= t^{AM} + t^{Map} + t^{Shuffle} + t^{Reduce} \\ &= t^l + (t^l + s^i/d^o + t^m + s^o/d^i + s^o/d^o \\ &\quad + s^o/d^i) * n^w + (s^o * n^c)/b^i + t^{Reduce} \end{aligned} \tag{4.1}$$

Table 4.1 shows the notations used in our estimation algorithm. Equation 4.1 gives an evaluation of time consumption for a MapReduce job. The AM setup time can be expressed by the container launch time t^l. The execution time of Map tasks t^{Map} includes 5 sub-phases: setup, read, map, spill and merge. The setup sub-phase can be shown as the container launch time t^l. The read sub-phase can be calculated by the input data size s^i divided by the disk output rate d^o. The map

sub-phase is symbolized by t^m, which can be evaluated by history records. The spill sub-phase writes the intermediate data into disk, *i.e.*, s^o/d^i. The merge sub-phase is to read the spilled data back for merging and write the merged data into disk again, *i.e.*, $s^o/d^o + s^o/d^i$, if the intermediate data is too large to spill once. The above analysis of Map phase is to calculate one wave, then we multiple it by the number of waves (n^w). The shuffle phase is intermediate data size divided by the bandwidth in one wave, other waves are not considered because there are overlaps between the Map phase and Shuffle phase.

$$t_u = t^m * (n^m/n_u^m) \tag{4.2}$$

$$t_d = (t^l + t^m + s^o/d^i) * (n^m/n^c) + (s^o * n^c)/b^i \tag{4.3}$$

Our algorithm to estimate the performance of the U+ and D+ modes are shown in Equations 4.2 and 4.3, respectively. Since we only consider one Reduce task, its execution time for both U+ and D+ modes are exactly the same, which can be omitted in Equations 4.2 and 4.3. The submission framework also removes the AM setup time (t^{AM}) from the Equation 4.1 for both modes. The setup sub-phase and Shuffle phase can be ignored due to a single container for the U+ mode. As the intermediate data are cached instead of dumping them into disk in the U+ mode, the time consumption of Spill and Merge, *i.e.*, $s^o/d^i + s^o/d^o + s^o/d^i$, are trivial. n^m/n_u^m shows the calculation of the number of waves for U+ mode. The overall performance t_u is calculated as Equation 4.2. The evaluation of time consuming for the D+ mode is shown in Equation 4.3. For a short job, a majority of Map tasks only spill to disk once, we can ignore the Merge sub-phase *i.e.*, $s^o/d^o + s^o/d^i$, and

Table 4.2: Microsoft Azure instance types

Instance Type	Cores	Memory	Disk	Price
A1	1	1.75GB	70GB	$0.09/hr
A2	2	3.5GB	135GB	$0.18/hr
A3	4	7GB	285GB	$0.36/hr

only consider the Spill sub-phase, *i.e.*, s^o/d^i. The Shuffle phase can be computed as $(s^o * n^c)/b^i$ due to the overlap between the Map phase and Shuffle of two adjacent waves. At last, the decision maker compares the execution time for the U+ mode and D+ mode to kill the slower one.

4.4 Experiments

The experiments were conducted on Microsoft Azure [54], which supports different types of servers such as A1, A2, A3. These instances differ in the number of cores, memory size, disk size, and price, as shown in Table 4.2.

Our experiments were performed on two different clusters, which have the same usage charge per hour. One cluster consists of 1 NameNode and 4 DataNodes of A3 instances. The another cluster consists of 1 NameNode and 9 DataNodes of A2 instance nodes. Each node runs CentOS Linux Server 7, JDK version 1.7, and Apache Hadoop version 2.2. To evaluate our optimization techniques, we ran three different benchmark applications from Hadoop example package, *i.e.*, WordCount, TeraSort, and PI. WordCount is a MapReduce program that counts words in input files. TeraSort samples the input data generated by TeraGen, and uses MapReduce to sort them into a total order. PI is a MapReduce program that estimates pi using a quasi-Monte Carlo method.

4.4.1 Experimental Results on A3 Cluster

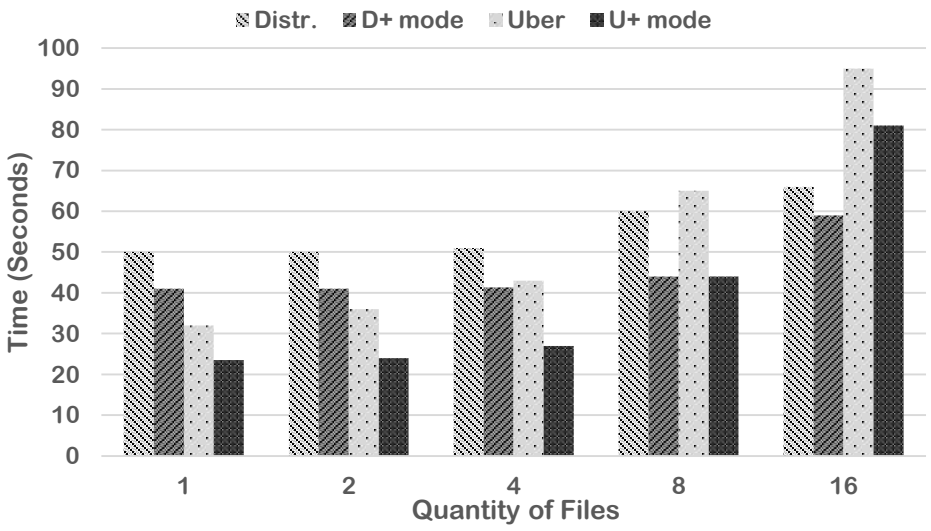


Figure 4.6: WordCount performance when varying the number of files but fixing the file size to 10MB.

In this experiment, we create a cluster consisting of 1 NameNode and 4 DataNodes of A3 instances. In Figure 4.6, the number of files varies from 1 to 16, and the size of each file is 10MB. We execute the WordCount benchmark to compare the performance under the original Hadoop and our MRapid. Our D+ mode gains an improvement of 36.36% compared to the original Hadoop when the file size is 8. The reason is that our improved scheduler chooses DataNodes that are relatively idle and have better data locality to allocate Map tasks. Our new submission framework also reduces the setup and allocation overhead of AMs. When the number of input files is 4, our U+ mode improves the performance by 59.26% compared to the original Uber mode. This is due to our parallel computing mechanism, which can execute Map tasks in parallel and avoid spilling intermediate data into disk when they are small. Figure 4.6 shows that when the number of in-

put files is very large, the D+ mode can only gain performance improvement by the submission framework, since the original Hadoop behaves nearly to our D+ mode in terms of data locality and resource usage. When the total input file size is 160 MB, the U+ mode has to spill intermediate data into the disk, which is similar to the original Uber mode, the improvement is 11.43%. From our experiment, when the number of files is 8, the D+ mode and the U+ mode have similar performance; when the number is more than 8, the U+ mode performs worse, even though it is still better than the original Uber mode.

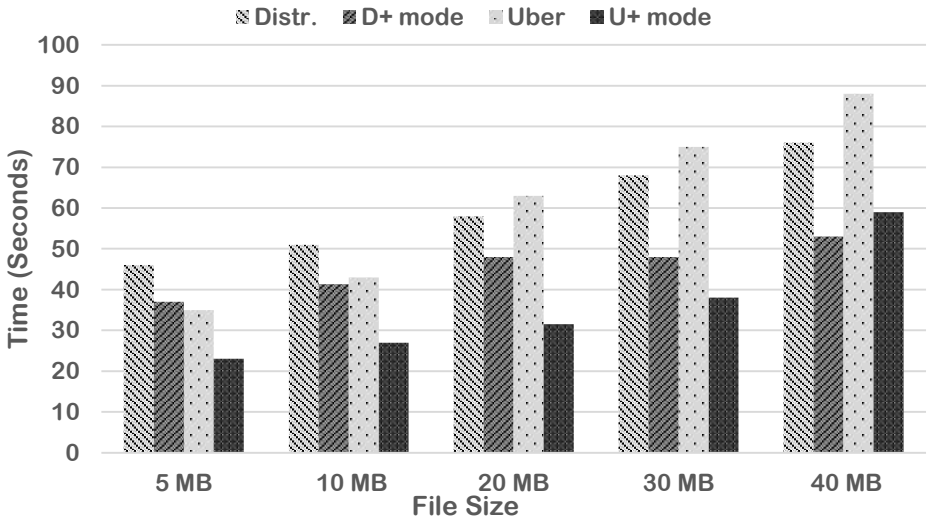


Figure 4.7: WordCount performance by fixing the number of files to 4 but varying file size.

Figure 4.7 demonstrates the performance of Hadoop and MRapid with 4 input files but the file size varies from 5 MB to 40 MB. The D+ mode can outperform the original Distributed mode by 43.40% when the file size is 40 MB, which is also 11.32% faster than the U+ mode. We observe that the D+ mode gains more performance improvement on larger file size. This is because Algorithm 4.1 schedules Map tasks as uniformly as possible; however, the original Hadoop only employs the resource of recently reported nodes, which can cause serious allocation imbalance for short jobs.

The performance of the D+ mode is better than the U+ mode when the total input data size is large, as the D+ mode can use cluster resource more efficiently than the U+ mode, which uses only one container to execute all tasks.

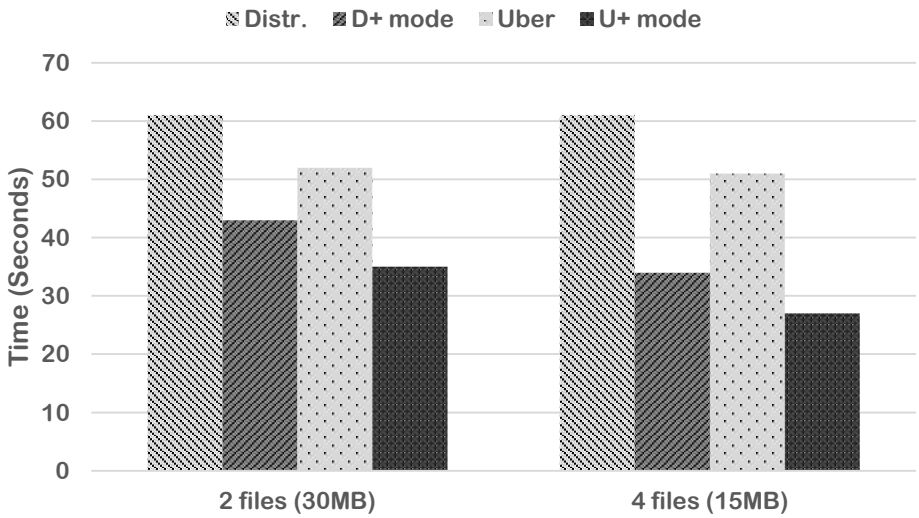


Figure 4.8: WordCount performance when fixing the input size to 60 MB.

Figure 4.8 shows the performance when the total file input size is fixed to 60 MB, and the number of files varies from 2 to 4. The performance of 4 files with the file size 15 MB is the best for the D+ mode, where we achieve 79.41% improvement due to better parallelism. The performance of the U+ mode is better when the number of files is 4 due to multithreading parallelism, which outperforms the original Uber mode by 88.89%.

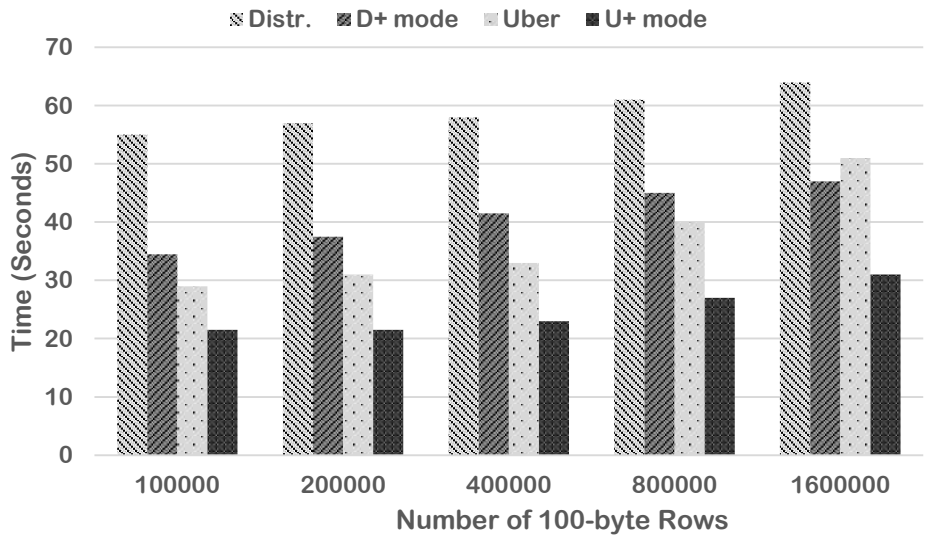


Figure 4.9: TeraSort performance with different numbers of rows.

Figure 4.9 shows the performance of another benchmark called TeraSort. We vary the number of 100-byte rows from 100k to 1,600k with 4 blocks, which designates 4 Map tasks. When the number of rows is 100k, the D+ mode gains 59.42% improvement compared to the original Hadoop. We also observe that the U+ mode is always better than the D+ mode; specifically, the U+ mode outperforms by 67% when the number of rows is 800k because such kind of applications do not require massive computation, and one container can handle it. We notice that the benchmark PI has the similar property, as shown in Figure 4.10. We vary the random number size from 100m to 1,600m for benchmark PI. When the random number size is more than 200m, for the original Hadoop, it is better to run PI in the original Distributed mode rather than the original Uber mode. However, for MRapid, when the random number size is large, *e.g.*, 1,600m, the U+ mode is still the better choice, which indicates that MRapid alleviates the limitation of the original Uber mode.

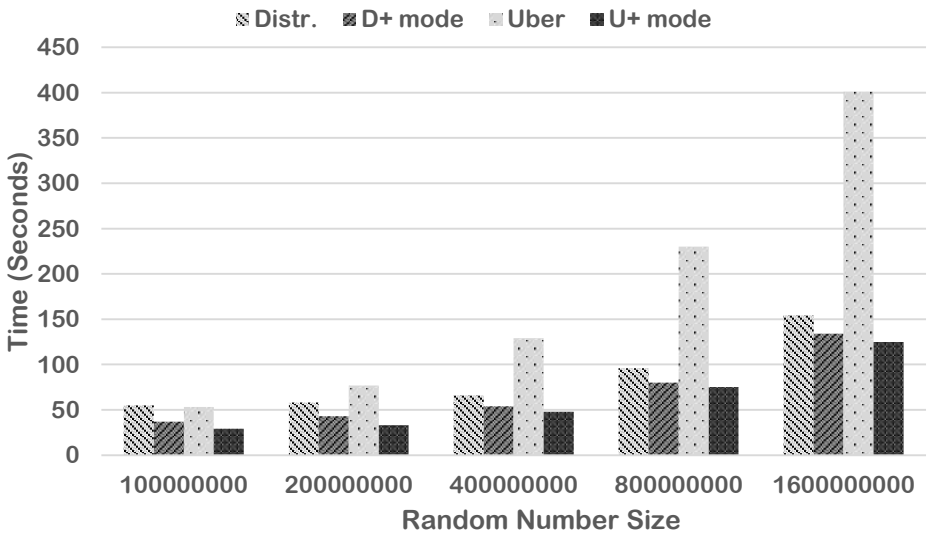


Figure 4.10: PI performance when varying the number of seeds.

4.4.2 Experimental Results for A2 Cluster

In this section, we discuss our experiments on another cluster consisting of 1 NameNode and 9 DataNodes of A2 instances.

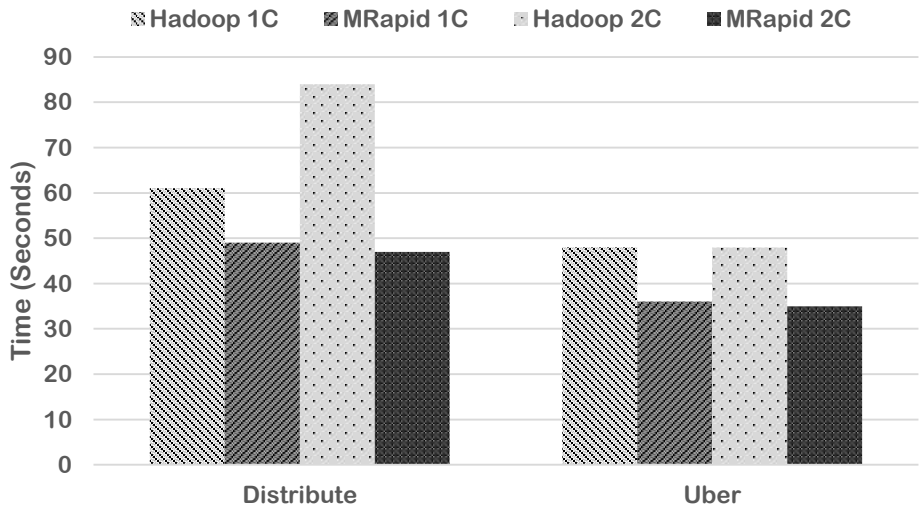


Figure 4.11: WordCount performance when varying the number of containers for each core.

Figure 4.11 compares the performance of our system with the original Hadoop when the number of containers allocated for each core is varied from 1 to 2 in A2 cluster. We find that the performance for MRapid does not fluctuate obviously, especially for the U+ mode when executing WordCount with four 10MB files. This is because the U+ mode only uses one container, and the D+ mode usually selects the relatively idle nodes to launch Map tasks. But for the original Hadoop, when the number of containers per core is 2, the performance of the original distributed mode becomes much worse due to its greedy scheduling.

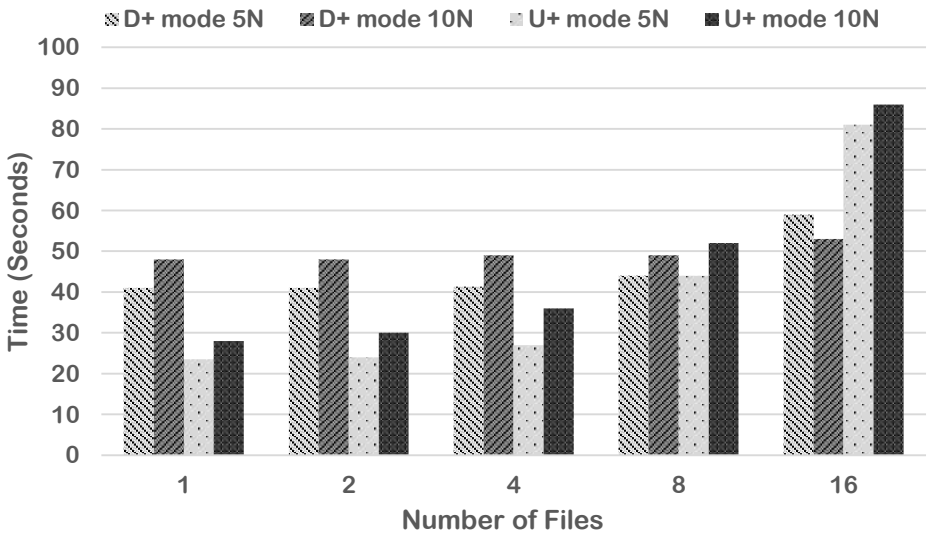


Figure 4.12: WordCount performance with different numbers of nodes.

For public cloud users, the cluster cost is often a concern. We compare the performance for a 10-node A2 cluster and a 5-node A3 cluster, which have around the same cost. As shown in Figure 4.12, for the U+ mode, it is always better to select the A3 cluster. For D+ mode, if the number of files is few and the cluster is relatively idle, it is better to use A3 cluster rather than A2 cluster; otherwise, it is better to deploy A2 cluster. The reason is that although the U+ mode just uses one container to execute the short job, if there are more resources available on the same node, the container for the U+ mode may steal these resources if allowed. For the D+ mode, if the number of files is large, such as 16 in Figure 4.12, although a cluster with more nodes at the same cost degrades the capability of each node, disk I/O and network contentions could be reduced; thus a cluster with more nodes may read input data and shuffle map results more efficiently.

4.5 Conclusions

In this work, we introduce an optimized Hadoop system to improve the performance of short jobs in two modes: D+ mode and U+ mode. In D+ mode, we design a new scheduler to schedule Map tasks based on the resource distribution situation and data locality. Instead of waiting for heartbeats reported from NMs to decide how to schedule tasks, our scheduler can allocate resources immediately. Our algorithm not only avoids allocation imbalance problem for short jobs, but also reduces the communication cost. For the U+ mode, rather than executing Map tasks sequentially, we employ multi-threading to run Map tasks in parallel. In addition, we cache the intermediate data into memory instead of writing them into disk and reading them later when the intermediate data are small. Moreover, we implement a new job submitting framework and speculative execution system to reduce the setup cost for short jobs. Our experiments show that our system can obtain significant performance improvement by 11% to 88% compared with the original Hadoop for short jobs.

CHAPTER 5: TUNING PERFORMANCE FOR BIG-DATA PLATFORM

In this work, we propose an efficient performance optimization engine called Hedgehog [1] to evaluate the performance based on "Law of Diminishing Marginal Utility" and give an optimal configuration setting. The initial experiments show that our optimization can gain 19.6% performance improvement compared to the naive configuration by tuning only 3 parameters.

5.1 Hedgehog Structure

Figure 5.1 shows the detailed architecture of our performance optimization engine, called Hedgehog. Hedgehog is a comprehensive performance tuning measure for Spark, which profiles the fine-grained executing information from Spark system, collects necessary information from Spark original logs, analyzes the workflow of Spark application, employs Detective Marginal Utility Model (DMU) to tune ratios among different memory categories, and optimize the performance by reasonable resource allocation. Hedgehog contains several components to coordinate the optimization for Spark.

[1]The content in this chapter was in part reproduced from the following article: Hong Zhang, Zixia Liu, Liqiang Wang, SMARTH: Enabling Multi-pipeline Data Transfer in HDFS, IEEE International Conference on Cloud Engineering, 2018. The copyright form for this article is included in Appendix C

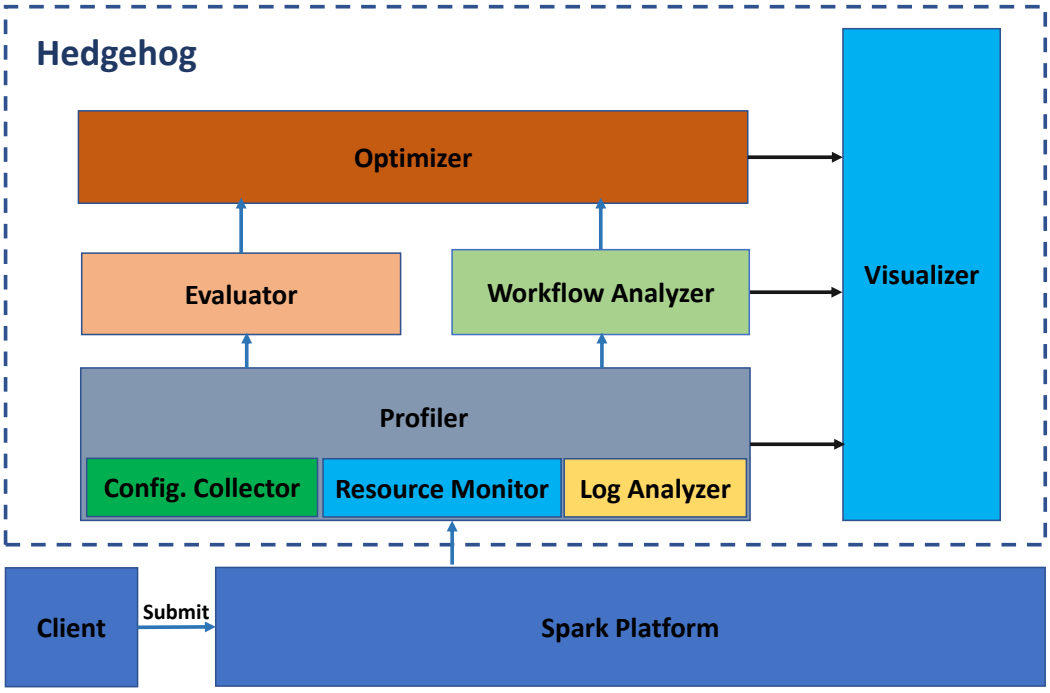


Figure 5.1: System Architecture of Hedgehog.

(1) **Dynamic Resource Monitor**: To monitor the system resource, including CPU usage, memory usage per task dynamically, our resource monitor records CPU usage and Java heap size for each Spark operation, including transformation, and action. The major difference between our monitor and others is that it can monitor the whole life cycle of each operation to exactly tell the resource utilization for each operation and identify the most influential operations in every stage.

(2) **Profiler**: As aforementioned, Spark application contains 5-level structure; however, the current execution profiling in Spark is far from being sufficient for more advanced performance model. We design a fine-grained profiler to collect more information to help users understand what is happening, provide higher prediction accuracy, and improve job performance, such as real-time data flow, data locality status, container status.

(3) **Configuration Collector**: Even being solely insufficient to our performance model, in reality, Spark already collects lots of useful information about the application, such as job, stage, task duration, and sub-phases like serialization/deserialization time and shuffle read/write time. This collector helps us collect all logs generated by Spark into a user-defined directory. Together with information collected by our profiler, we are able to collect all information from the fine-grained level to highest level.

(4) **Log Analyzer**: We design a log analyzer to extract necessary information, and organize them with the application structure by logs of Spark and our profiler.

(5) **Evaluator**: Based on the information collected, we design a brand-new resource-based performance model to predict application duration. Since Spark utilizes in-memory processing, which keeps data in-memory as much as possible to improve the performance, the influence of the resource especially the memory is very important. A reasonable resource allocation and configuration not only improves the performance of each task, but also optimizes the parallelism to speed up the execution for each stage.

(6) **Workflow Analyzer**: The major functionality of the workflow analyzer is to detect recursive structures in the application. Each stage has a description to describe its major operations, invoked class and code line number. Stages with same description reveal the executing of same task. With scanning all stages in ascending order, we use the stage description as key, the latest stage ID as value, and calculate the stage interval between the neighboring stages with the same key. Then we calculate iterations for this application to simplify the evaluation process.

(7) **Optimizer**: With the knowledge obtained from the evaluator and workflow analyzer, the optimizer employs "Law of Diminishing Marginal Utility" to allocate the memory resource properly. The major configuration problem for user is to configure the executor size (CPU cores, and memory size). Our optimizer can analyze the real CPU usage and memory usage for different phases to

improve the resource utilization ratio.

5.2 Performance Model

Since there are over 150 parameters in Spark. we cannot determine their affection exhaustively. Some parameters have significant impact on performance, some of them depend on the job characteristics. Some of them must be set before running a job, some could be changed after a job launches but before the task runs, and some of them could be reset while the task is running. Hence, figuring out the affection and types of these parameters is very important for our performance model and tuning mechanism. In our performance model, we only consider those general, important, but tricky-tuning parameters, especially related to memory.

$$t_{app} = \sum_{i=1}^{n} t_{job}^{i} = \sum_{i=1}^{n} \sum_{j=1}^{i_m} t_{stage}^{ij} \tag{5.1}$$

The total execution time t_{app} can be calculated by Equation 5.1, t_{job}^{i} denotes the duration of job i, and the duration of stage j in job i is symbolized by t_{stage}^{ij}. The execution time of each stage can be calculated by Equation 5.2. The whole stage is divided into 3 sub-phases: read, run, and write, their durations are t_{read}, t_{run}, and t_{write}, respectively. Spark needs to spill the intermediate data into disk and merge them later if the data are too large. Thus the sub-phases of read and write contain *spill* and *merge* processes, which are related to the execution memory. The read sub-phase also needs to load data from the previous stage or from file system, therefore it has *load* process, which is tied with the storage memory. The user memory affects the duration of the garbage collection process t_{gc}, which belongs to the *run* sub-phase. Because of the limitation of space, we omit details

here.

$$
\begin{aligned}
t_{stage} &= t_{read} + t_{run} + t_{write} \\
&= t_{rload} + t_{rspill} + t_{rmerge} \\
&+ t_{exact} + t_{gc} + t_{wspill} + t_{wmerge}
\end{aligned}
\tag{5.2}
$$

5.3 Experiments

We evaluated Hedgehog on a cluster containing 10 nodes, 1 NameNode and 9 DataNode. Each node has an Intel(R) Xeon(R) CPU E5-2620 v3 with 6 cores, and 32 GB memory. Our Spark cluster is based on CentOS Linux Server 7, JDK version 1.8, Apache Hadoop version 2.7 and Apache Spark 2.1. We build a very simple performance model which only bound 3 configuration parameters: "executor-memory", "executor-cores", and "num-executors". The results show that our optimizer can gain 19.6% performance improvement compared to the naive configuration.

5.4 Conclusions

In this paper, we design an end-to-end performance model for Spark, and use 3-level sampling model to sample the test application, *i.e.*, input data level, task level, and element level. We also introduce a new white box performance model to evaluate the performance based on "Law of Diminishing Marginal Utility". Initial results show that our optimization can gain 19.6% performance improvement compared to the naive configuration, even by tuning only 3 parameters.

CHAPTER 6: AN ADAPTIVE STOCHASTIC GRADIENT DESCENT ALGORITHM ON BIG-DATA PLATFORM

In this work, we optimize the current implementation of SGD in Spark's MLlib by reusing data partition for multiple times within a single iteration to find better candidate weights in a more efficient way. Whether using multiple local iterations within each partition is dynamically decided by the 68-95-99.7 rule. We also design a variant of momentum algorithm to optimize step size in every iteration. This method uses a new adaptive rule that decreases the step size whenever neighboring gradients show differing directions of significance. Experiments show that our adaptive algorithm is more efficient and can be 7 times faster compared to the original MLlib's SGD.

6.1 Background

Gradient descent is an iterative optimization algorithm that minimizes an error function defined by a set of parameters. We use the terms weight and parameter interchangeably in this paper. There are several steps in finding a local minimum of a function using gradient descent: (1) Initialize weights with random values. (2) Compute the gradient, which is the first derivative of the error function $err(w)$, since it is the fastest decreasing direction for the current weights. (3) Update the weights with the negative gradients, as shown in Equation 6.1, where γ is the learning rate. (4) Repeat steps 2 and 3 until the error cannot be reduced notably or already reach the maximum iterations.

$$W_t = W_{t-1} - \gamma \nabla err(W_{t-1}) \tag{6.1}$$

However, calculating gradients over the entire training set is often too expensive, and may have a high probability of obtaining a partial optimal solution. Moreover, if the entire dataset is too large to be cached in memory, performance can degrade significantly. Stochastic gradient descent (SGD), on the other hand, is a stochastic approximation of the gradient descent algorithm by using a few training examples or a minibatch from the training set to update parameters in every iteration. It avoids the high cost of calculating gradients over the whole training set, but is sensitive to feature scaling. It can be denoted as follows.

$$W_t = W_{t-1} - \gamma \sum_{i=1}^{n} \nabla err_i(W_{t-1})/n \tag{6.2}$$

where n is the number of samples in a minibatch.

Stochastic gradient descent is one of the most important optimizers in Spark MLlib. Algorithm 6.1 shows the process of calculating stochastic gradient descent in Spark MLlib. At first, it broadcasts the initial weights or the weights calculated by the previous iteration to every compute node, which may host one or more partitions of datasets. Then the input RDD is sampled according to the minibatch rate. After that, it calculates gradient for each sample in every partition, and aggregates all gradients by the driver using a multi-pass tree-like reduce (which is called *TreeAggregate*). At the end of each iteration, the weights are updated according to Equation 6.2. This process terminates when all iterations are executed or when it has converged.

There are three major problems of SGD in Spark: (1) the data uploaded are only used once in each iteration. If the memory reserved for this application is not large enough to cache all data, it must read data from disk or even worse from other compute nodes. (2) *treeAggregate* operation reduces all gradients by multiple stages, which is inefficient. (3) The original learning rate updating

algorithm is too simple, which makes convergence too slow.

Algorithm 6.1 Parallel SGD of Original MLlib

Input: $input\_rdd, init\_weights, num\_iterations, minibatch\_rate$

Output: $weights$

```
1:  weights = init_weights
2:  for step = 1 to num_iterations do
3:      broadcast(weights)
4:      for all partitions in input_rdd parallel do
5:          samples = partition.sample(minibatch_rate)
6:          for each sample in samples do
7:              grad = computeGradient(weights, sample)
8:          end for
9:      end for
10:     grads = tree aggregate gradients from all compute nodes
11:     weights = updateWeights(weights, grads)
12: end forreturn weights
```

Designing asynchronous parallel stochastic gradient descent algorithms with or without lock is an active area in recent years. Liu *et al.* [55] introduce an asynchronous parallel stochastic descent algorithm that achieves a linear convergence rate and almost linear speedup on a multicore system. But it requires the cost function that satisfies an essential strong convexity property. AsySVRG [56] is an asynchronous SGD variant (SVRG) that adopts a lock-free strategy and convergent with a linear convergence rate. But this algorithm is only designed for multicore systems, which has some limitations to deploy on clusters of multiple machines. HOGWILD! [57] implements SGD in parallel that allows processors to overwrite each other's work without locking, but the gradient updates only modify small parts of the weights to avoid conflicts. Zinkevich *et al.* [58] present

a novel data-parallel stochastic gradient descent algorithm to reduce I/O overhead but must place every sample on every machine. However, these algorithms cannot be applied to Spark because Spark collects parameters from all compute nodes using low-frequent synchronous methods like *reduce* and *aggregate* rather than the high-frequent pushing and pulling mechanism used by the aforementioned algorithms.

A few approaches have been proposed to improve the performance of calculating SGD based on model parallelism and data parallelism. Zhang *et al.* [59] design an asynchronous SGD system on multiple GPUs working together to calculate gradients and update the global model parameters. However, when extending it to a multi-server multi-GPU architecture, the performance becomes poor due to the network bottleneck. DistBelief [60] is a software framework that introduces two algorithms, Downpour SGD and Sandblaster, for large-scale distributed training using tens of thousands of CPU cores. GPU A-SGD [61] is a new system that makes use of both model parallelism and data parallelism which is similar to DistBelief [60] but with GPUs to speed up training of convolutional neural networks. However, none of these system is compatible with Spark framework because all of them need intensive communications through a centralized parameter server, which is hard to be implemented efficiently on Spark.

To improve the training efficiency of gradient descent, there are several projects to develop adaptive learning rate techniques. Jacobs [62] analyzes why the steepest descent is slow to converge and propose four heuristics to speed up the convergence. Simple adaptive momentum (SAM) [63] dynamically adjusts the momentum-coefficient by the similarities between the current weights and previous weights to reduce the negative effect of overshooting the target. [64] introduces a fast convergent algorithm based on Fletcher-Reeves update by adaptively changing the gradient search direction. Unfortunately, none of them is implemented distributively on a cluster, and have no guarantee of the convergence.

There are also some papers to discuss the optimizations based on platforms like Hadoop and Spark or some specific hardwares [65, 66]. Zhang *et al.* [8] employ a caching technique to avoid disk I/O for short jobs. HogWild++ [67] is a novel decentralized asynchronous SGD algorithm which replaces the global model vector with a set of local model vectors on top of multi-socket NUMA systems. [68] and [69] discuss how to build a computing framework to support Large-scale applications like Logistic Regression and Linear Support Vector Machines.

6.2 Design and Implementation

To overcome the problems mentioned above, we design a novel algorithm that has inner iterations within each global iteration. An inner iteration updates local weights multiple times without sending back the gradients before the accumulated weights are reduced on the driver at the end of each global iteration. This approach dramatically reduces the amount of communication and avoid aggregating gradients in multiple stages.

Algorithm 6.2 Parallel SGD with Iterated Local Search

Input: $input\_rdd, init\_weights, global\_iters, minibatch\_rate, local\_iters$

Output: g_ws

1: $g\_ws = init\_weights$
2: **for** $g = 1$ to $global\_iters$ **do**
3: $broadcast(g\_ws)$
4: **for all** $partitions$ in $input\_rdd$ **parallel do**
5: $l\_ws = g\_ws$
6: **for** $l = 1$ to $local\_iters$ **do**
7: $samples = partition.sample(minibatch\_rate)$
8: $l\_ws = computeWeights(l\_ws, samples)$
9: **end for**
10: **end for**
11: $weights$ = aggregate l_ws from all compute nodes
12: // Test if weights satisfy 68-95-99.7 rule
13: **if** $(g == 1$ && $!satisfyGaussianDist(weights))$ **then**
14: $local\_iters = 1$
15: **end if**
16: $g\_ws = averageWeights(weights)$
17: **end forreturn** g_ws

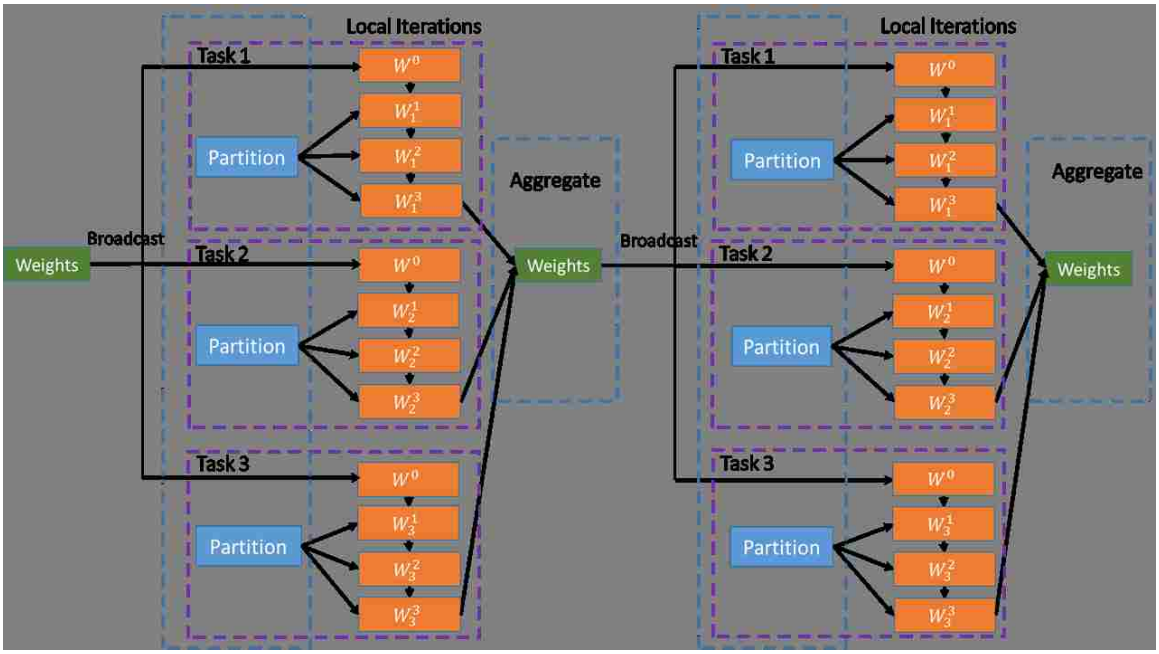


Figure 6.1: Gradient Descent with Iterative Local Search.

6.2.1 Parallel SGD with iterative local search

Algorithm 6.2 shows the parallel SGD algorithm with local iterations. Within each global iteration, we asynchronously update local weights multiple times within local iterations before updating the global weights using the new average weights at the end of each global iteration on the driver. In each local iteration, the calculated weights are used to compute subsequent weights using the same partition. Let *local_iters* denote the number of local iterations, which is determined by the features of input data, computational complexity of algorithm, and capability of cluster. Roughly, we notice that the *local_iters* is inversely proportional to the standard deviation of the weights calculated from all computers to guarantee convergence. *local_iters* is a hyper-parameter in our current system. It will be our future work to investigate how to find the optimal *local_iters* adaptively.

In the first global iteration, we check whether the weights calculated by all partitions fit a Gaussian distribution. If so, our optimizations are applied; otherwise, we fallback to the original algorithm in Spark's MLlib due to having too significant difference among the calculated weights on all partitions. Note that in Algorithm 6.2, weights are aggregated by the driver. However, in Algorithm 6.1, gradients are aggregated by the driver. Such an optimization is based on Spark's intrinsic features. In the original MLlib implementation shown in Algorithm 6.1, a partition is sampled once and each sample generates a vector of gradients. All these gradients within one global iteration are reduced to the driver based on tree-aggregation. In our Algorithm 6.2, all gradients are applied to the local weights, and only the weights are transfered at the end of each global iteration. This approach dramatically reduces the data to be transferred. In addition, tree-aggregation consists of multiple stages, which may be efficient for large-scale Spark systems, but could degrade the performance of small-scale systems.

Because of the limit of pages, we have no space to give the proof of convergence which depends on the distribution of the weights.

6.2.2 68-95-99.7 Rule

As mentioned above, we need to know the distribution of our data to determine whether to employ aggressive local iterative search to speedup. However, to assert the input data as normal is more complex and time consuming, like Kolmogorov-Smirnov test (K-S test) [70]. From the analysis of the convergence, we found that 99.7% of vectors of weights calculated by all compute nodes lie within 3σ, and that σ is small enough. So we use the 68-95-99.7 rule instead of hypothesis testing

to check whether the sets of weights collected from all compute nodes have a normal distribution.

$$similarity = \frac{A \bullet B}{||A|| \times ||B||} \tag{6.3}$$

This empirical rule is the facts that 68.27%, 95.45% and 99.73% of the values in a normal distribution fall within one, two and three standard deviations of the mean, respectively. Since one set of weights is a vector, to simplify the process, we employ cosine similarity [71] between each set of weights and the mean weights to check whether the weights satisfy the 68-95-99.7 rule. Equation 6.3 shows the measure of similarity between two non-zero vectors.

6.2.3 Parallel SGD with Adaptive Momentum

Spark MLlib, by default, uses a simple adaptive updater that adjusts the learning rate by the inverse of the square root of the number of iterations executed, as shown in Equation 6.4. This algorithm does not include any heuristics proposed by [62]. It only reduces the step size gradually to ensure the convergence for non-convex optimization problems.

$$w_t = w_{t-1} + \frac{current\_step}{\sqrt{num\_steps}} w_{t-1} \tag{6.4}$$

As one of the heuristics indicated by [62], Momentum adds a fraction of the previous updating vector to the current gradients, as shown in Equation 6.5. If the previous updating vector is in the same direction with the current gradient, it increases the step size towards the target; otherwise the

step size is reduced. But this algorithm causes an overshooting problem, *i.e.*, the search step strides over the minima but the next search direction does not turn round since the momentum term is too large.

$$\begin{aligned} v_t &= \alpha v_{t-1} + \beta \nabla err(w) \\ w_t &= w_{t-1} - v_t \end{aligned} \tag{6.5}$$

Swanston [62] proposes a simple adaptive momentum (SAM) algorithm that still uses constant terms for both learning rate and momentum, but adds a coefficient to adaptively adjust the momentum according to the similarities between the last gradient and the current gradient. If two adjacent gradients have similar directions ($\cos(\theta) > 0$), it increases the influence of the previous iteration; otherwise, it reduces the influence and changes direction quickly. But this algorithm neither analyzes the root cause of the oscillation nor gives a guideline for adjusting the learning rate.

Algorithm 6.3 shows the steps of calculating SGD in parallel with adaptive momentum. At the beginning of each global iteration, we broadcast the current average weights *g_ws* to all compute nodes. Then for each partition, we calculate a vector of weights by local iterations. In each local iteration, we sample the partition randomly with the *minibatch* rate. Then we calculate the gradients for each sample with the present local weights. After that, we check cosine similarity between the previous gradient and the current gradient. If the cosine value is less than 0, which means the current direction is totally different from the previous direction, there must be overshooting for some weights. There are two potential reasons causing this phenomenon: (1) the momentum term is too large, so even if the learning rate is small, the current step still strides over the target; (2) the learning rate is too large, and the weights cross over and are on the other side. Between the

two causes, we must decide which is the dominating factor for overshooting. Firstly, we adjust the momentum coefficient α to 0 to check whether the momentum is the root cause. Then we update the local weights, and start the next local iteration. If there is no oscillation in the next iteration, then it indicates that momentum being too large caused this oscillation. Otherwise, the oscillation must be caused by the learning rate being too fast.

We do not adjust the learning rate between local iterations, but simply terminate local iterations if oscillation is detected. Here we use *vibrate_last* to store whether or not there is an oscillation in the previous iteration. To summarize, if oscillation occurs, we first adjust the momentum coefficient to 0; and if two consecutive oscillations occur, we reduce the learning rate by half.

$$\begin{aligned} v_t &= \alpha(1+\cos(\theta))v_{t-1} + \beta\nabla err(w) \\ w_t &= w_{t-1} - v_t \end{aligned} \tag{6.6}$$

Algorithm 6.3 Parallel SGD with Adaptive Momentum

Input: $input\_rdd, l\_ws, old\_grad, local\_iters, \alpha, \beta, minibatch\_rate$

Output: g_ws

```
1:  g_ws = init_weights;
2:  for g = 1 to global_iters do
3:      broadcast(g_ws)
4:      for all parts in input_rdd parallel do
5:          oscillation = 0
6:          vibrate_last = false
7:          for l = 1 to local_iters do
8:              samples = part.sample(minibatch_rate)
9:              new_grad = computeGrad(l_ws, samples);
10:             similarity = cosθ(old_grad, new_grad)
11:             if (similarity < 0) then
12:                 if (vibrate_last) then
13:                     oscillation = 1
14:                     break
15:                 else
16:                     vibrate_last = true
17:                     l_ws = momentum(0, β, old_grad, new_grad)
18:                 end if
19:             else
20:                 vibrate_last = false
21:                 l_ws = momentum(α, β, old_grad, new_grad)
22:             end if
23:         end for
24:     end for
25:     num_oscils = aggregate oscillation from all nodes
26:     if (num_oscils/num_parts >= 0.5) then
27:         β = β/2.0
28:     end if
29:     weights = aggregate l_ws from all compute nodes
30:     if (g == 1 && !satisfyGaussianDist(weights)) then
31:         local_iters = 1
32:     end if
33:     g_ws = averageWeights(weights)
34: end forreturn g_ws
```

6.3 Experiments

Our experiments were performed on a cluster consisting of 1 NameNode and 6 DataNodes. Each node has an Intel(R) Xeon(R) CPU E5-2620 v3 with 6 cores, and 32 GB memory. Our Spark cluster is based on CentOS Linux Server 7, JDK version 1.8, Apache Hadoop version 2.7 and Apache Spark 2.1. We use a representative gradient descent method, Linear Regression, as benchmark to test the performance of FTSGD.

6.3.1 Experiments without Adaptive Learning Rate

We first compare the performance between the original SGD in MLlib and Algorithm 6.2 without considering the effect of the optimization of adaptive learning rate.

Figure 6.2 shows the accuracy of different number of local iterations, which is evaluated by mean squared error (MSE). In this experiment, the total input data size is 50 GB, which are generated by `LinearDataGenerator` class in MLlib package. Each sample has 100 features, and the scaling factor ε is 0.1. The initial learning rate is 1.0. When the local iteration is 1, our algorithm is the same as MLlib's SGD. When the number of local iterations is 6, we find that the accuracy is the best. It only use two global iterations to converge MSE to a very small value (1.029), which is even better than the accuracy of MLlib's SGD with 15 iterations.

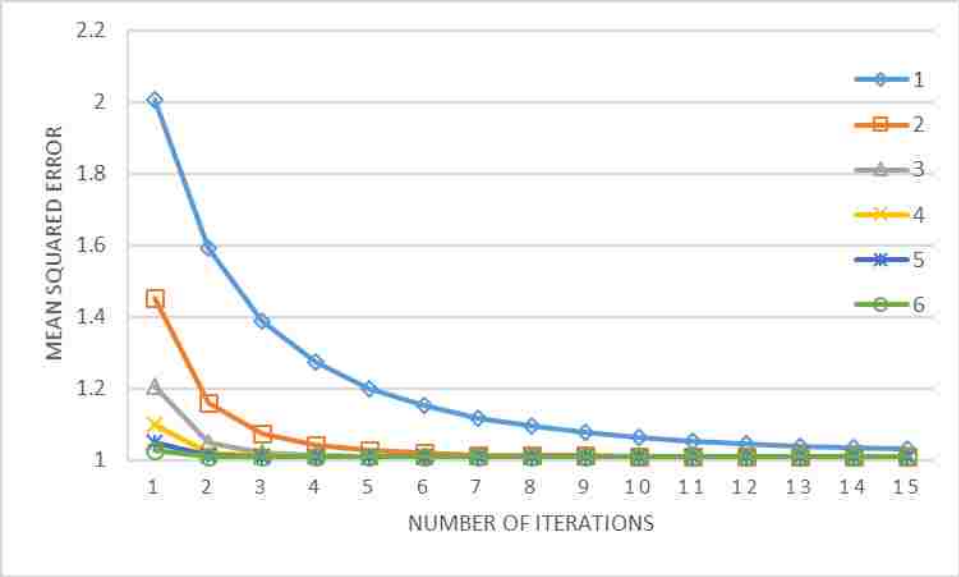


Figure 6.2: Accuracy of different numbers of local iterations in every global iteration.

Figure 6.3 indicates the performance of different number of local iterations with the same configuration in Figure 6.2. Although each global iteration in our algorithm is slower than MLlib's SGD, the performance of our algorithm is still better. This is because the convergence of our algorithm is fast, and we avoid the communication time of tree aggregation. When the number of local iterations is 5, the performance of our algorithm is best, which is 6.5 times faster than MLlib SGD with the same MSE. Although FTSGD with 6 local iterations is more accurate than with 5 iterations using the same global iteration, the longer execution time for each global iteration undermines the performance improvement.

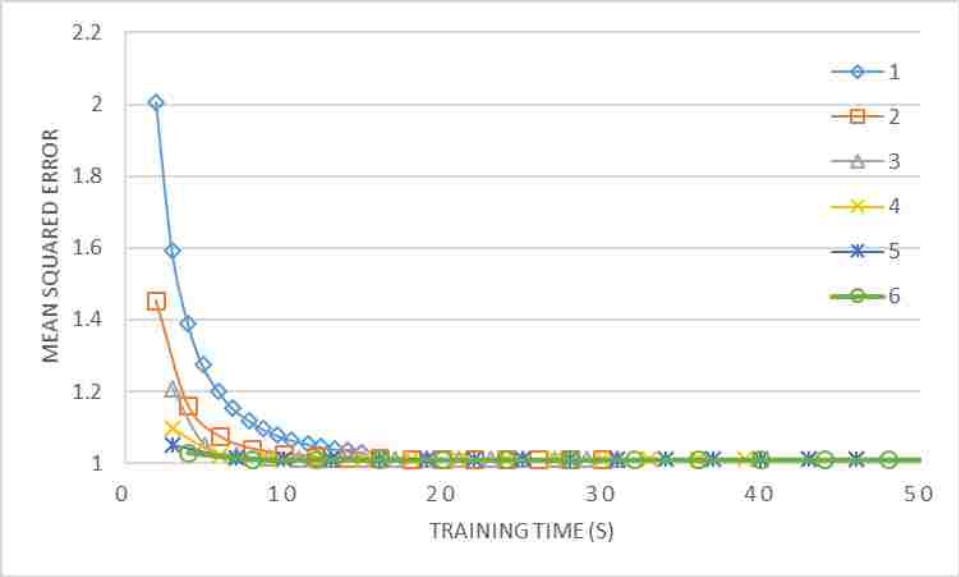
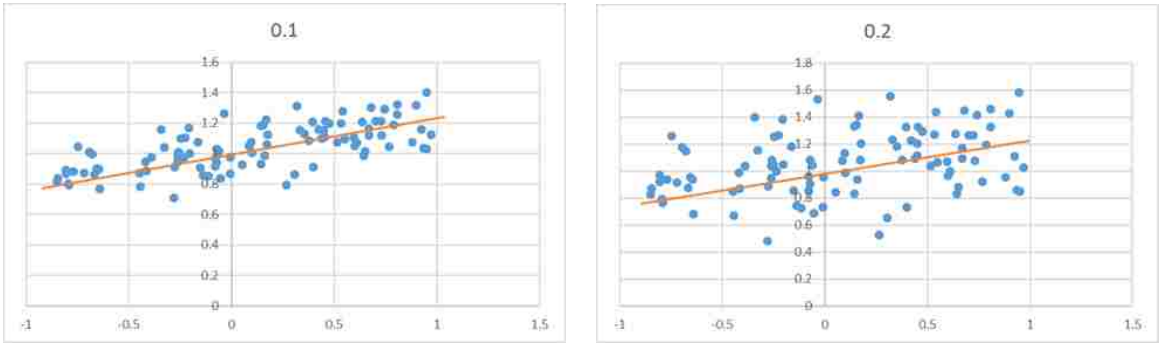


Figure 6.3: Performance of different numbers of local iterations.

Figure 6.4 demonstrates the input datasets with different ε scaling factor. The ε scaling factor is used to add white Gaussian noise to a full linear dataset. Figure 6.4(b) is the dataset with scaling factor 0.2, which is wider than the dataset in Figure 6.4(a) with scaling factor 0.1. Figure 6.5 shows the convergence with datasets of different scaling factor. We find that even with a large scaling factor 0.5, our algorithm with 3 local iterations is still convergent. This means if the distribution of data is an uniform normal distribution, our algorithm can converge, even the standard deviation is a little bigger. It is reasonable that the larger scaling factor dataset has larger MSE.



(a) Scaling factor: 0.1 (b) Scaling factor: 0.2

Figure 6.4: Data distributions with different scaling factor.

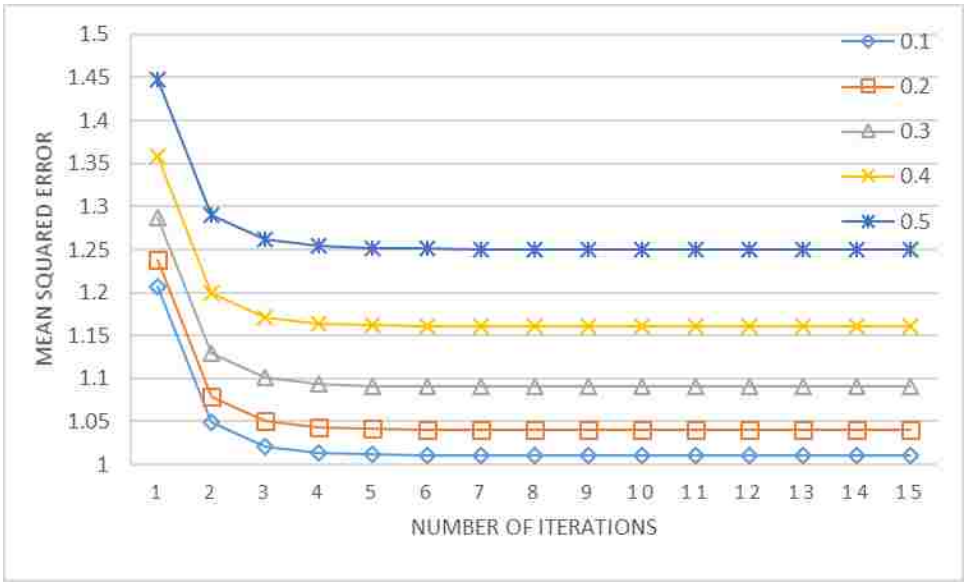


Figure 6.5: Accuracy of different scaling factor

6.3.2 *Experiments with Adaptive Learning Rate*

In this experiment part, we discuss the performance of our algorithm with adaptive learning rate (*i.e.*, Algorithm 6.3 or FTSGD).

Figure 6.6 compares the performance between Algorithm 6.2 and Algorithm 6.3. We find that when the number of global iterations is 5, the accuracy of FTSGD is better than Algorithm 6.2 that uses the learning rate updating algorithm in Equation 6.6. Another is that the MSE difference between two adjacent iterations is very small after 10 global iterations (less than 0.00001), which means that FTSGD is convergent when the global iteration is 10, and FTSGD can terminate earlier than Algorithm 6.2 without learning rate optimization.

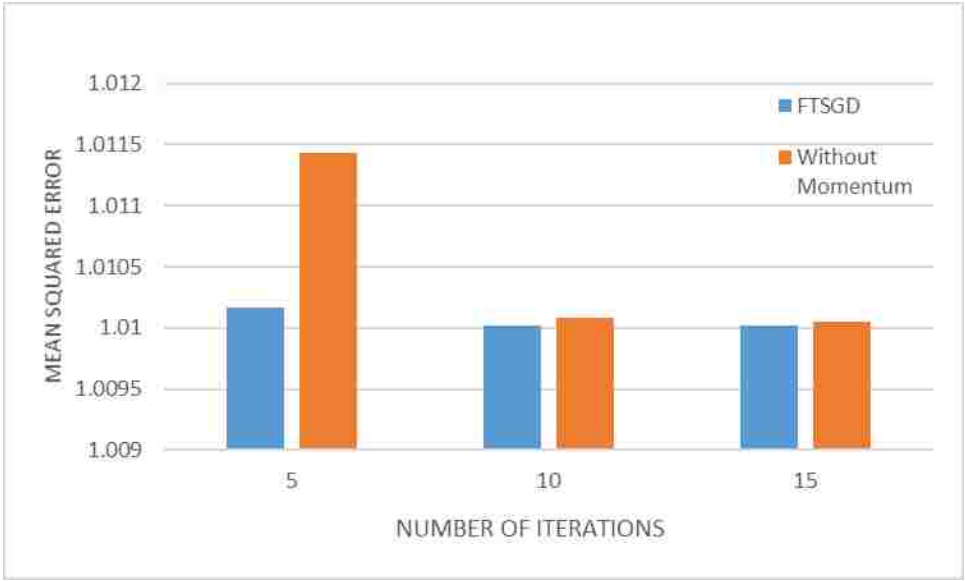


Figure 6.6: Performance with and without adaptive learning rate.

Figure 6.7 shows the performance of FTSGD with different input data sizes. We vary the input data size from 25 GB to 100 GB. When the input data size is 25 GB and 50 GB, FTSGD can outperform

the original SGD by about 4.3 times and 3.6 times, respectively. When the input data size is larger than 50 GB, the cost increases exponentially. This is due to data size exceeding memory size, which forces disk I/Os because data cannot be cached in memory. There are three kinds of tasks: (1) process local, where the data are cached in local memory; (2) node local, where the data are stored in local disk; (3) rack local, where the data must be fetched from a remote node in the same rack. We have 100 GB memory in our cluster, and 50% is reserved for OS and Spark system. If input is larger than 50 GB, Spark has to read data from disk (node local) or remotely (rack local). It is very common that input data are too large to be cached entirely into memory. That is why we reuse the data loaded in memory to do an asynchronous updating to save execution time. Figure 6.7 demonstrates that when the input data size is 100 GB, FTSGD only spends 198 seconds to converge and gives a better accuracy compared to MLlib's SGD that takes 1401 seconds.

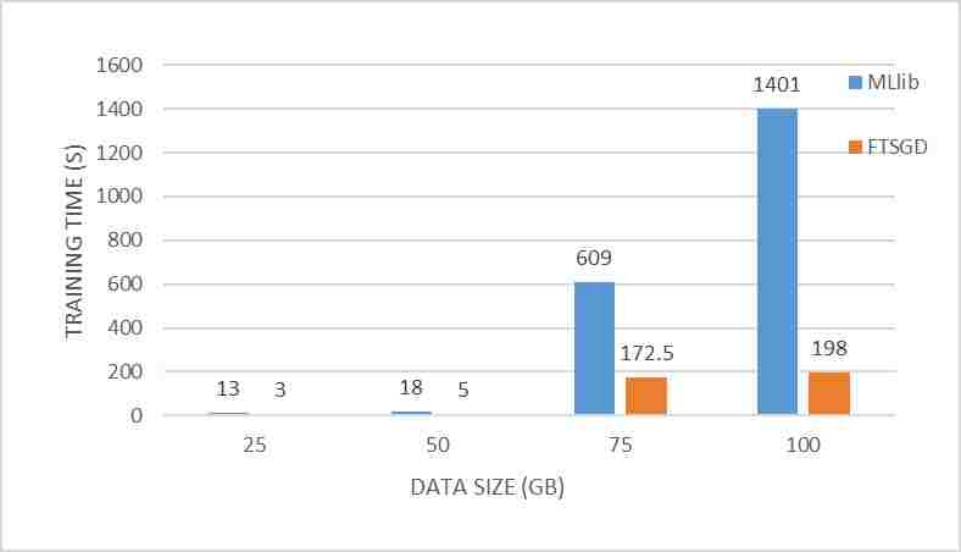


Figure 6.7: Performance of different input data sizes.

Figure 6.8 shows the performance of FTSGD with the initial learning rate varied from 0.01 to 1.0. We notice that FTSGD with initial learning rate 1.0 gains more performance improvement. From

Figure 6.8, the initial learning rate cannot be too small since a small rate may cause too many steps towards the optimal target. However, the initial learning rate cannot be too large since a large value will cause overshooting problem, even non-convergence. Because FTSGD can reduce the learning rate quickly if there exists oscillation, we can set the initial learning rate slightly larger, which also can detect more area to avoid stepping into local optimum.

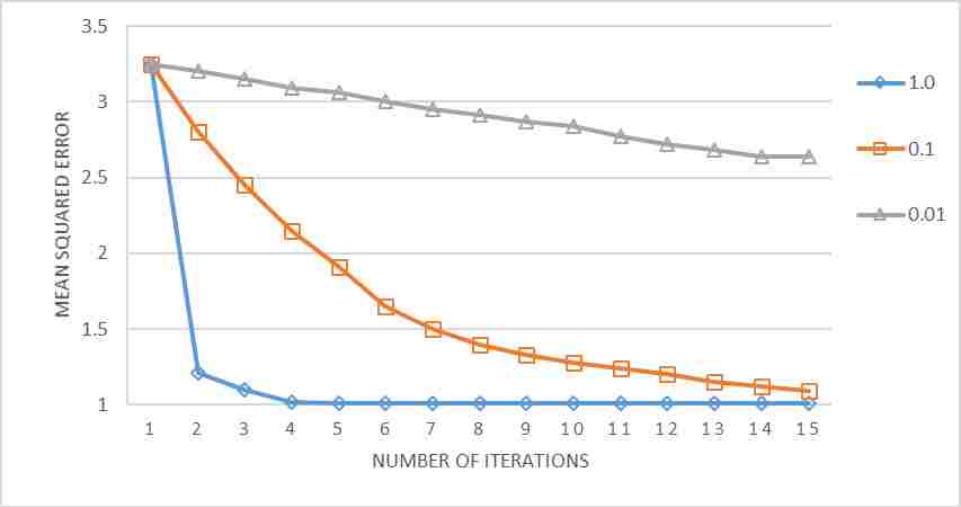


Figure 6.8: Performance of different initial learning rate

Figure 6.9 demonstrates the effect of the minibatch rate. It is obvious that the larger the minibatch rate is, the longer it takes for one global iteration. For 50 GB input data size, it takes 22 seconds to calculate one global iteration when setting minibatch 0.2, but 42 seconds for minibatch 0.8, which almost doubles cost. But the accuracy is similar since the input data is very uniform.

Table 6.1: Performance of two data set with different ratios

	1:1	2:1	3:1	4:1	5:1	10:1	20:1	100:0.3
Mean	0	0.004329004	0.00654233	0.007792208	0.008658009	0.010625738	0.011750155	0.012909324
STDEV	0.012987013	0.012244273	0.011218753	0.01038961	0.009679948	0.007467007	0.005531399	0.001418401
68-95-99.7 Rule	No	No	No	No	No	No	No	yes
MSE (MLlib)	1.763563134	1.593385182	1.507113571	1.451839211	1.414957992	1.344010868	1.293488292	1.258641468
MSE (5 Local Iters)	1.829216647	1.615403542	1.518092382	1.458397027	1.418999539	1.345111833	1.293792029	1.258606695

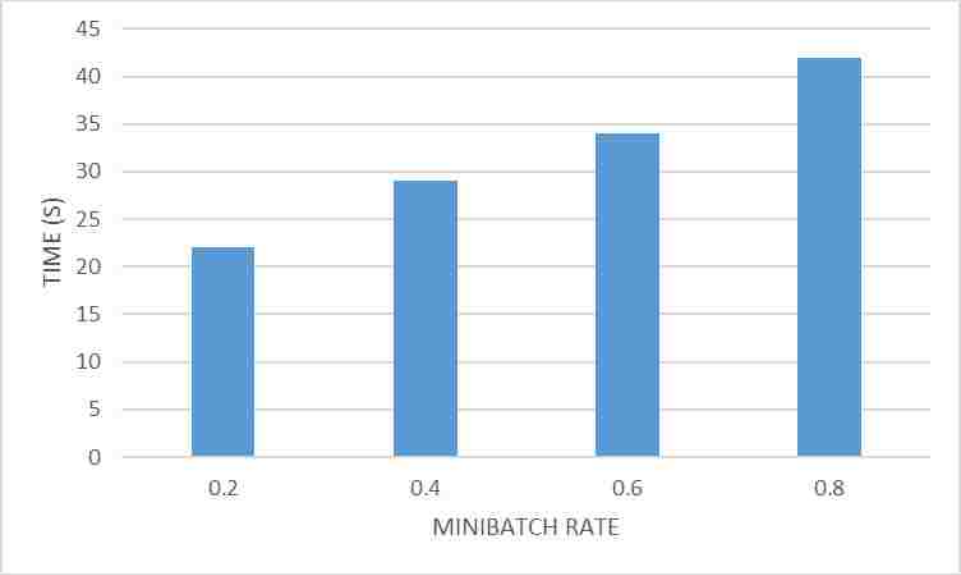


Figure 6.9: Performance of different minibatch rate.

In order to test the performance when the input data is non-uniform, we generate a data set which contains two data distributions, as shown in Figure 6.10. We vary the ratio of the number of samples in two datasets, and the results are shown in Table 6.1 for 15 iterations. In Table 6.1, we calculate mean and standard deviation, and check if the parameters satisfy the 68-95-99.7 rule. We compare the performance of MLlib's SGD and FTSGD with 5 local iterations. The results shows that if the data is not a normal distribution, only the dataset satisfying the 68-95-99.7 rule can

outperform the original SGD algorithm in MLlib. Even the data size ratio between two datasets is large such as 20:1, MLlib's SGD still shows better performance than FTSGD if the 68-95-99.7 rule is not satisfied. However, when the data size ratio between two datasets reaches 100:0.3, the 68-95-99.7 rule is satisfied and our algorithm finally outperforms MLlib's SGD.

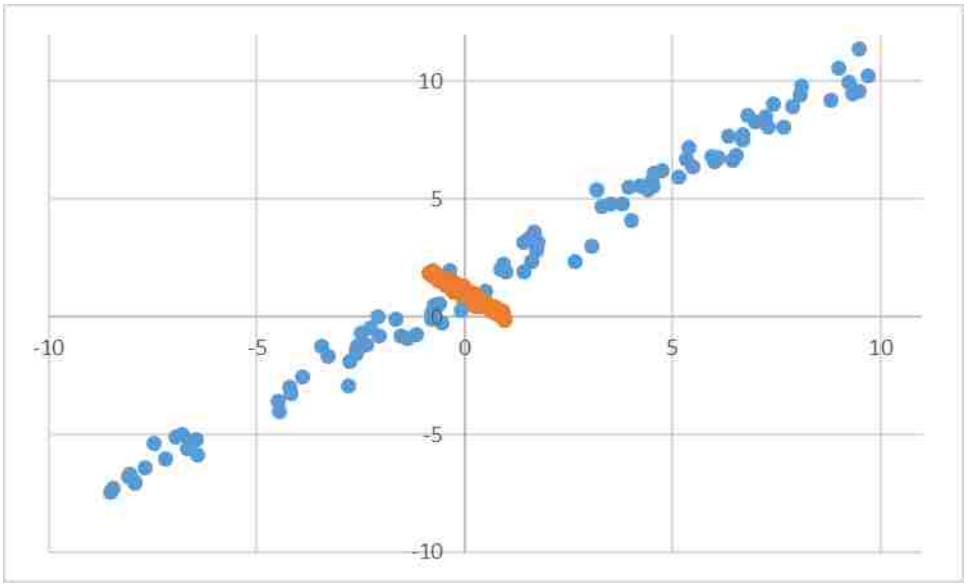


Figure 6.10: Two datasets with different distributions.

6.3.3 Experiments with Different Size of Partitions

In this experiment, we measure the performance of input datasets with different number of partitions. We vary the number of partitions of 50 GB data from 500 to 2000, then the partition size is changed from 120 MB to 30 MB. As shown by Figure 6.11, we notice that the larger the partition size is, the better the performance is, because the large partition size can reduce the total number of tasks, and then decrease the overhead of task setup and cleanup. The maximum partition size is bounded by the block size in HDFS (128 MB in our cluster).

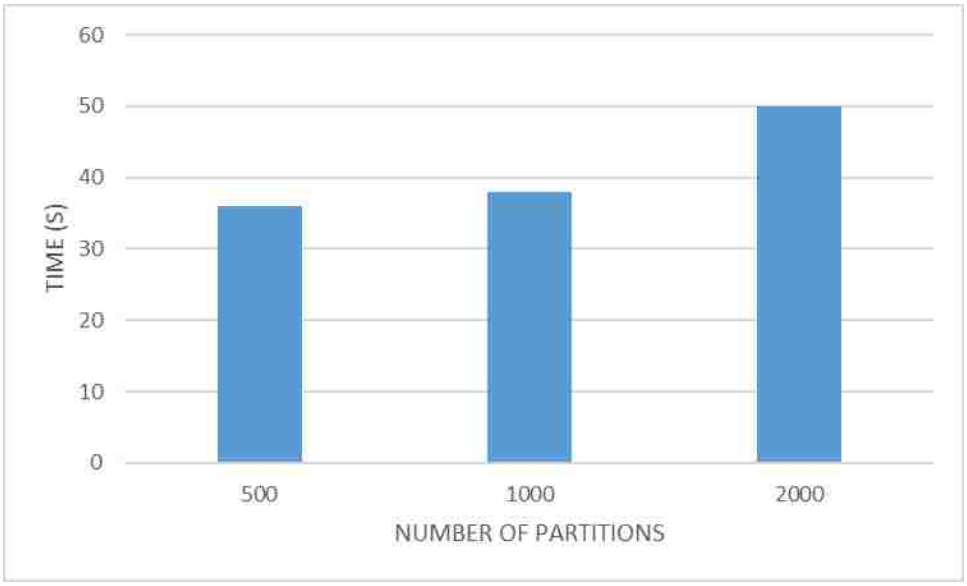


Figure 6.11: Performance of different number of partitions with the same data size.

6.4 Conclusions

In this work, we design an asynchronous parallel SGD algorithm with iterative local search, called FTSGD, by reusing data partition multiple times within a single global iteration. We also design a variant of momentum algorithm to find the optimal step size on every iteration, which uses a new adaptive rule to decrease the step size whenever the neighboring gradients have been shown in different directions. The experiments show that our algorithm is more efficient and reaches convergence faster than the MLlib library.

CHAPTER 7: AN EFFICIENT GEOSPATIONAL AND TEMPORAL ANALYTICS FRAMEWORK

This work introduces a scalable and distributed geographic information system, called Dart [1], based on Hadoop and HBase. Dart provides a hybrid table schema to store spatial data in HBase so that the Reduce process can be omitted for operations like calculating the mean center and the median center. It employs reasonable pre-splitting and hash techniques to avoid data imbalance and hot region problems. It also supports massive spatial data analysis like K-Nearest Neighbors (KNN) and Geometric Median Distribution. In our experiments, we evaluate the performance of Dart by processing 160 GB Twitter data on an Amazon EC2 cluster. The experimental results show that Dart is very scalable and efficient.

[1]The content in this chapter was in part reproduced from the following article: Hong Zhang, Zhibo Sun, Zixia Liu, Chen Xu, Liqiang Wang, Dart: A Geographic Information System on Hadoop, IEEE 8th International Conference on Cloud Computing, 2015. The copyright form for this article is included in Appendix D

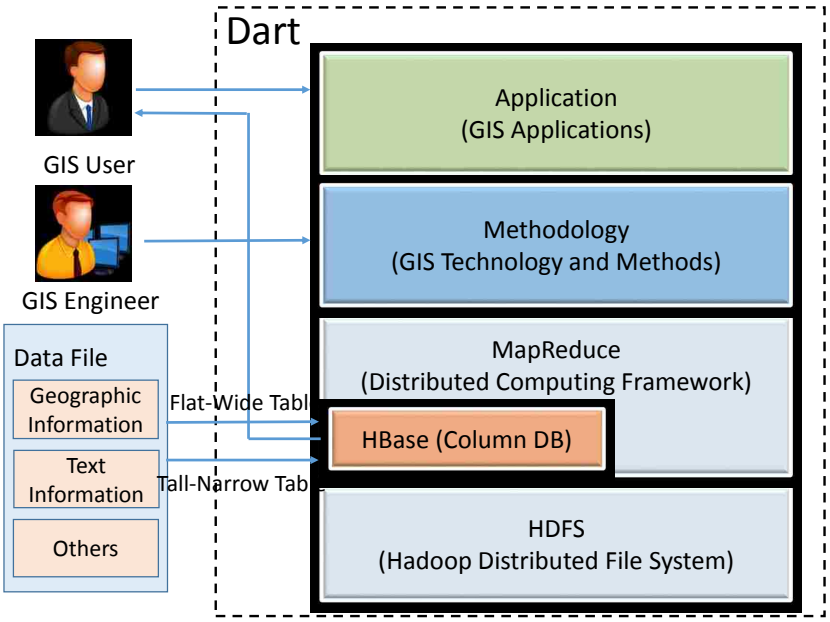


Figure 7.1: System architecture of Dart.

Figure 7.1 shows an outline of our spatial analyzing system Dart for social network. Our system can automatically harvest Twitter data and upload them into HBase. It decomposes data into two components: the geographic information, and the text information, then insert them into tall-narrow and flat-wide tables, respectively. Our system targets two types of users: GIS engineers and GIS users. GIS engineers can make use of our system to develop new methods and functions, and design additional spatial modules to provide more complex data analysis; GIS users can build complicated data analysis models based on current spatial data and methods for tasks such as analyzing the geographic mean and median centers. All input and output data are stored in HBase to provide easy and efficient search. It also supports a spatio-temporal search for fine-grained data

analysis.

Dart consists of four layers: computing layer, storage layer, methodology layer, and data analysis layer. The computing layer offers a MapReduce computing model based on Hadoop. The storage layers employs a NoSQL database, HBase, to store spatio-temporal data. We design a hybrid table schema for data storage, and use pre-splitting and uniform hashes to avoid data imbalance and hot region problems. Our implementation on Hadoop and HBase has been optimized to process spatial data from social media like Twitter. The methodology layer is to carry out complex spatial operations such as figuring out the geographic median or facilitate some statistical treatments to support the upper data analysis layer. In the data analysis layer, Dart supports specific analytics like KNN and geometric median distribution from customers.

7.2 Methodology

In this section, we describe how to calculate the geographic mean, midpoint, and median. We present a brand-new algorithm for the geometric median calculation that (1) starts the iteration with a more precise initial point and (2) imposes a grid framework to the process to reduces the total iteration steps. These three geographic indicators are important estimators for summarizing location distribution patterns in GIS. For example, it could help us estimate a person's activity space more accurately.

$$\begin{aligned} Lat &= \sum_{i=1}^{n} lat_i/n \\ Lon &= \sum_{i=1}^{n} lon_i/n \end{aligned} \tag{7.1}$$

7.2.1 Geographic mean

The main idea of the geographic mean is to calculate an average latitude and longitude point for all locations. The projection effect in mean center calculation has been ignored in this study because the areas of daily activity space of Twitter users are normally small. Equation 7.1 shows basic calculation steps . The main problem here is how to handle points near or on both sides of the International Date Line. When the distance between two locations is less than 250 miles (400 km), mean is approximate to the true midpoint[72].

7.2.2 Geographic midpoint

The geographic midpoint (also known as the geographic center, or center of gravity) is the average coordinate for a set of points on a spherical earth. If a number of points are marked on a world globe, the geographic midpoint is at the geographic center among these points.

Initially, latitude and longitude of each location are converted into three dimensional cartesian coordinates after changing unit to radians . We then compute the weighted arithmetic mean of cartesian coordinates of all locations (use 1 as weight by default). After that, the three dimensional average coordinate is changed back to latitude and longitude in degrees.

7.2.3 Geographic Median

To calculate the geographic median of a set of points, we need to find a point that minimizes the total distance to all other points. A typical problem here is the Optimal Meeting Point (OMP) problem that has considerably practical significance. The implementation of this algorithm is more complex than the geographic mean and the geographic midpoint since the optimal point is

approached iteratively.

Algorithm 7.1 The original algorithm for median

Input: Location set $S = \{(lat_1, lon_1), ..., (lat_n, lon_n)\}$

Output: Coordinates of the geographic median

Let *CurrentPoint* be the geographic midpoint
$MinimumDistance =$
    $totalDistances(CurrentPoint, S)$
for $i = 1$ to n **do**
    $distance = totalDistances(location_i, S)$
    if $(distance < MinimumDistance)$ **then**
        $CurrentPoint = location_i$
        $MinimumDistance = distance$
    end if
end for
Let TestStep be diagonal length of the district
while $(TestStep > 2 \times 10^{-8})$ **do**
    $updateCurrentPoint(CurrentPoint, TestStep)$
end whilereturn CurrentPoint

Figure 7.1 shows the general steps of the original algorithm. Let *CurrentPoint* be the geographic midpoint computed above as the initial point, and let *MinimumDistance* be the sum of all distances from *CurrentPoint* to all other points. In order to find a relatively precise initial iteration point, we count the total distance from each place to other places; if any of these places has a smaller distance than *CurrentPoint*, replace *CurrentPoint* by it and update *MinimumDistance*. Let *TestStep* be $PI/2$ radians as the initial step size, then generate eight test points in all cardinal and intermediate directions of the *CurrentPoint*. That is, to the north, northeast, east, southeast, south, southwest,

west and northwest of the *CurrentPoint* with the same distance of $TestStep$. If any of these eight points has a smaller total distance than *MinimumDistance*, make this point as *CurrentPoint* and update *MinimumDistance*. Otherwise, reduce $TestStep$ by half, and continue searching another eight points around *CurrentPoint* by the new $TestStep$ until $TestStep$ meet the precision (2×10^{-8} radians by default).

We can compute the distance using the spherical law of cosines. If distances between each pair of points are small, we then use distance of spatial straight line between two points to replace circular arc. The equation of the spherical law of cosines is shown as follows.

$$\begin{aligned} distance = \arcsin(&\sin(lat_1) * \sin(lat_2) + \\ &\cos(lat_1) * \cos(lat_2) * \cos(lon_2 - lon_1)) \end{aligned} \tag{7.2}$$

Algorithm 7.2 The improved algorithm for greographic median

Input: Location set $S = \{(lat_1, lon_1), ..., (lat_n, lon_n)\}$

Output: Coordinates of the geographic median

1: Let $CurrentPoint$ be the geographic midpoint;
2: $MinimumDistance =$
3: $\quad totalDistances(CurrentPoint, S)$
4: Divide the district into grids;
5: Calculate the center coordinates for each grid and count the number of locations distributed in each grid as weight;
6: Calculate the total weighted distance between center of each grid to centers of other grids;
7: If any center has a smaller distance than CurrentPoint's, replace it with this center, and update;
8: Let TestStep be diagonal length of grid divided by 2;
9: **while** $(TestStep > 2 \times 10^{-8})$ **do**
10: $\quad updateCurrentPoint(CurrentPoint, TestStep)$
11: **end whilereturn** CurrentPoint

We observe that in order to select a better initial iteration point, the original algorithm has to compute distances between every couple of points. The time complexity of such selection procedure is $O(n^2)$. In fact, if we remove this procedure (from line 3 to line 9 in Algorithm 7.1), the time complexity for finding the initial point will be reduced to $O(n)$. It shows a significant improvement for total running time of calculating Geographic Median in our experiments compared to the original algorithm, especially when the point set is large (more than 100 points).

The time cost of the initial point detection procedure in the original algorithm outruns its performance improvement, which is the reason why it should be omitted. However, a good initial point selection schema can potentially decrease the number of iterations and ameliorate total perfor-

mance. Thus, we design a brand-new algorithm that employs grid segmentation to decrease the cost of searching a proper initial point, which can also reduce the initial step size at the same time. Its efficiency and performance improvement are demonstrated by our experiments. As shown in Algorithm 7.2, after computing the geographic midpoint, we partition the district into grids with the same size. Since our algorithm reduces at least one iteration, in order to search a better initial point, we take the expense of at most one iteration to select a better initial point from centers of grids. We count the number of points located in each grid as weight, and calculate the total weighted distances from the center of each grid to centers of other grids. If any center is better than *CurrentPoint* in the sense of less total distance, replace *CurrentPoint*. As the centers of eight neighbor grids of *CurrentPoint* is not better than its own, we can reduce the initial step to half of distance between two diagonal centers.

Algorithm 7.3 MapReduce program for KNN

function: Map(k,v)

1: $p = context.getPoint()$
2: **for** each cell c in value.rawCells() **do**
3: **if** column family is "coordinates" **then**
4: **if** qualifier is "minipoint" **then**
5: $rowKey = c.row()$
6: $location = c.value()$
7: $distance = calculateDistance(location, p)$
8: **end if**
9: **end if**
10: **end for**
11: emit (1, *rowKey* + "," + *distance*)

function: Combine, Reduce(k,v)

12: $K = context.getK()$
13: **for** (each v_i in v) **do**
14: $(rowKey_i, distance_i) = v_i.split(,)$
15: **end for**
16: Sort all users by distances, and choose K smallest distance locations to emit;

7.3 Data Analysis

In this section, we introduce two common spatial applications, namely, KNN and geometric median distribution. KNN is a method for classifying objects based on the closest training examples according to some metrics such as Euclidean distance or Manhattan distance. KNN is an important

module in social media analytics to help user find other nearby users. Geometric median distribution is to count users' distribution in different areas, which might be useful for business to promote products. Due to the mobility of users, the geographic median is one of the best values to stand for users' geographic positions.

7.3.1 K Nearest Neighbors

Algorithm 7.3 shows the process of MapReduce on Hadoop. In the map function, we extract *point* for KNN search and K value from *context*. Then we calculate the distance from each point to *point*, and send top K points to the combine function and then the reduce function. Both of them sort users by distances computed by the map function. The difference between the combine function and the reduce function is that the former sends results to the reduce task, but the later one uploads K nearest neighbors to HBase.

7.3.2 Spatial distribution

Algorithm 7.4 describes how to calculate the distribution of users in a district. In the map function, we obtain the grid length (0.01 degree by default) from *context* to build a mesh on the region of interest, and compute which grid each point locates in. Then the key-value pair of grid index and the number of users is sent to the combine function and afterwards reduce function. The combine and reduce functions sum the number of users in each grid, and finally upload results into HBase.

Algorithm 7.4 MapReduce program for geometric median distribution

function: Map(k,v)

1: $gridLength = context.getGridLength()$
2: **for** each cell c in value.rawCells() **do**
3: **if** column family is "coordinates" **then**
4: **if** qualifier is "minipoint" **then**
5: $(lat, lon) = c.value()$
6: $latIdx = (lat - miniLat)/gridLength$
7: $lonIdx = (lat - miniLon)/gridLength$
8: **end if**
9: **end if**
10: **end for**
11: emit $((latIdx, lonIdx), 1)$

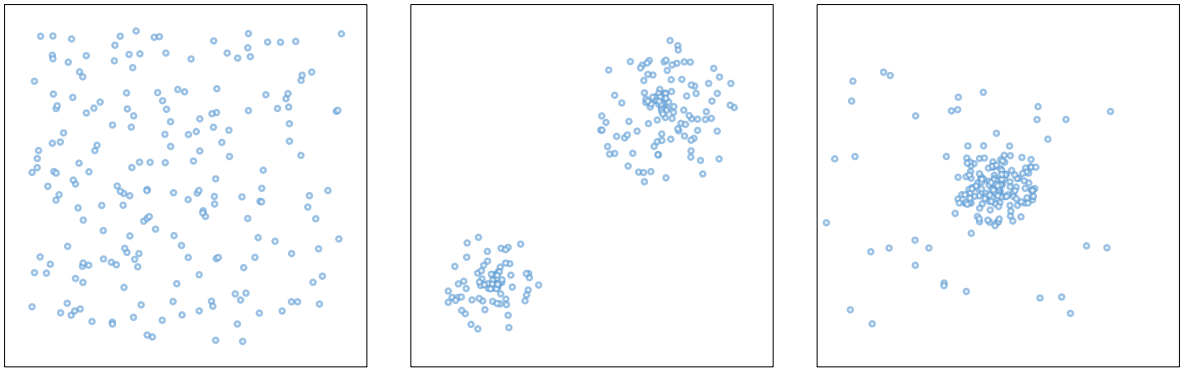
function: Combine, Reduce(k,v)

12: $total = 0$
13: **for** (each v_i in v) **do**
14: $total$ += v_i
15: **end for**
16: emit $(k, total)$

7.4 Experiments

In this section, we describe the experiments to evaluate the performance of Dart based on Hadoop and HBase, including the computations on mean center, median center, KNN, and geometric median distribution.

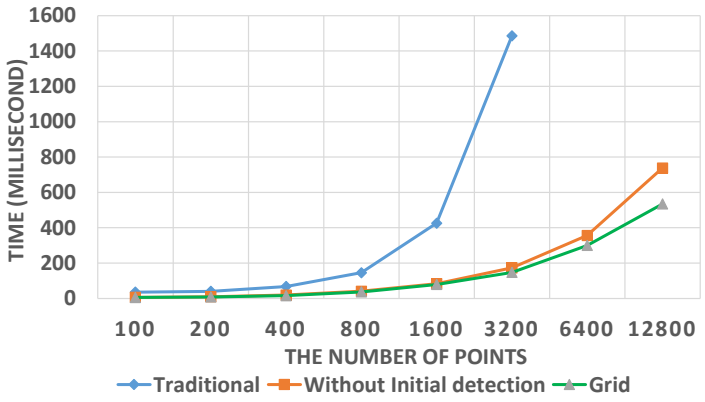
Our experiments were performed on Amazon EC2 cluster that contains one Namenode and nine Datanodes. Each node is an Amazon EC2 m3.large instance, which provides a balance of compute, memory, and network resources. EC2 m3.large instance has Intel Xeon E5-2670 v2 (Ivy Bridge) processors, 7.5 GB memory, 2 vCPUs, SSD-based instance storage for fast I/O performance, and runs CentOS v6.6. Our Hadoop cluster is based on Apache Hadoop 2.2.0, Apache HBase 0.98.8, and Java 6.



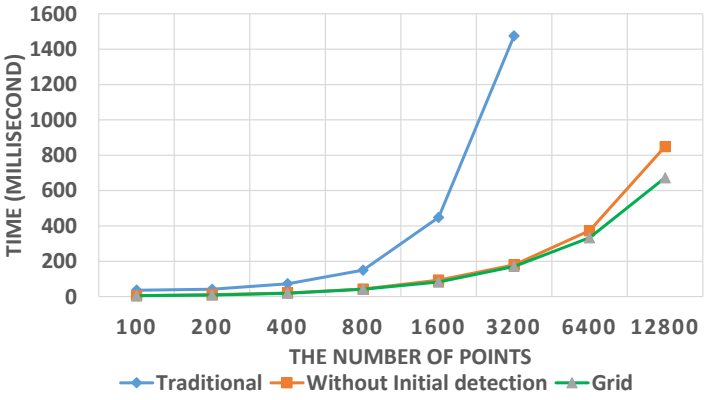
(a) uniform (b) two-area (c) skew

Figure 7.2: Data distributions.

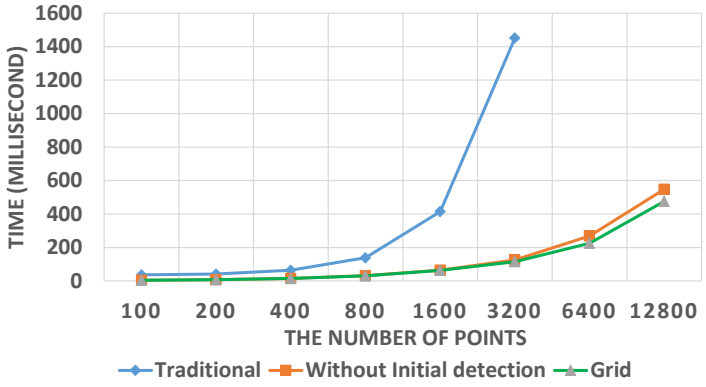
In our experiments, we extract Twitter data from 38 degrees North latitude and 73 degrees West longitude to 41.5 degrees North latitude and 77.5 degrees West longitude with a total size of 160 GB and more than 1 million users. This area mainly includes the metropolitan areas from New York City, Philadelphia, to Washington DC. In order to show the performance of our algorithm on a single machine, we generate random points to simulate three common scenarios shown in Figure 7.2: (1) Uniform is a scenario where all points are scattered uniformly in the district; (2) Two-area is that all points are clustered into two groups, which is very common in real world. (3) Skew is a scenario where most of points gather together but few points scatter outside.



(a) uniform



(b) two-area



(c) skew

Figure 7.3: Performance comparison of calculating mean and median.

Figure 7.3 shows the time of calculating the geographic median in three different scenarios. These experiments were conducted on a single m3.large node in Amazon EC2 cluster, and the number of points vary from 100 to 12800. We evaluated three algorithms: (1) The original algorithm most commonly used in geography[72]. (2) The algorithm without-initial-detection, which removes the procedure of selecting a better initial point from the original algorithm as we mentioned in Section 7.2.3. This algorithm reduces the time complexity of selecting initial point from $O(n^2)$ to $O(n)$. (3) Our own grid algorithm, which utilizes grid technique to improve the accuracy of initial point and reduce the step size dramatically at a reasonable time cost. This algorithm reduces the overall iteration number. As Figure 7.3 shows, when the number of points increases from 100 to 12800, the performance of Dart increases gradually. In all scenarios, our algorithm can outperform the original one by 9 to 11 times when the number of points is 3200. When the number of points is 12800, our new algorithm in Dart gains an improvement of 38%, 26%, and 15% compared to the without-initial-detection algorithm in the uniform, two-area, and skew scenarios, respectively. We find that the without-initial-detection and the grid algorithms cost less time in the skew scenario compared to the uniform and two-area scenarios because the midpoint is closer to the median center.

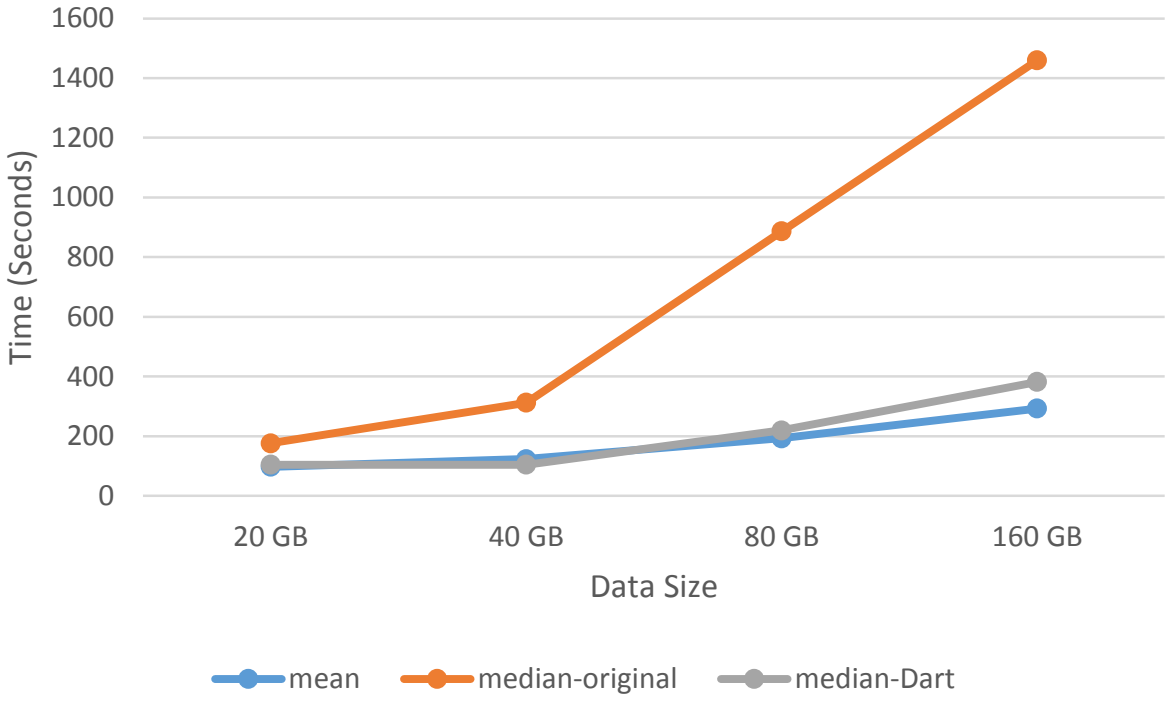
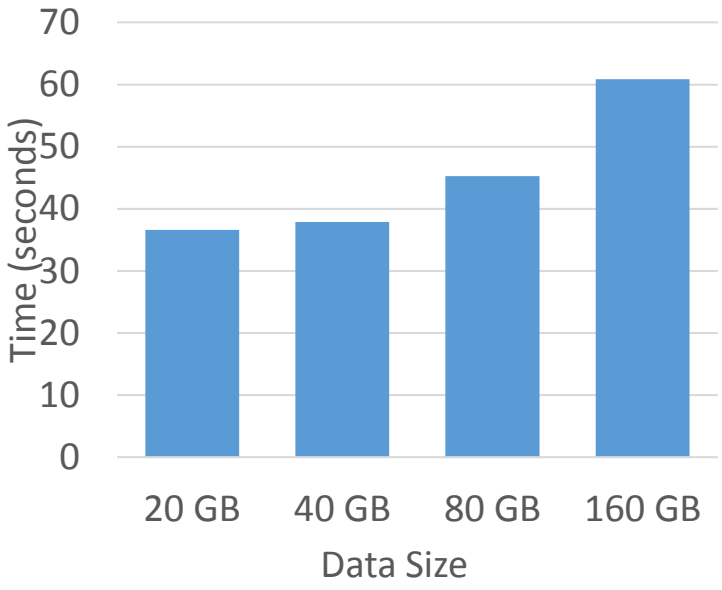
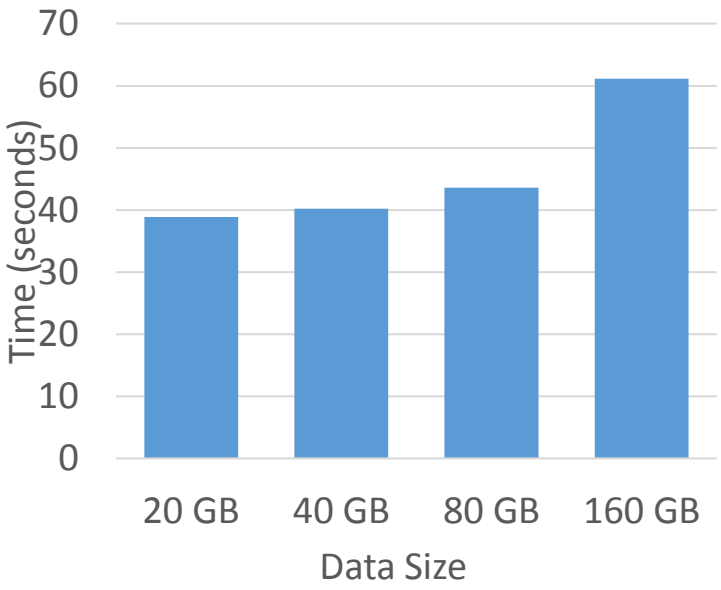


Figure 7.4: Performance comparison between mean and median.

Figure 7.4 shows the performance comparison of calculating the geographic mean and geographic median on Hadoop cluster, where the input data size varies from 20 GB to 160 GB. When the input data size is 160GB, our algorithm achieves an improvement of 7 times compared to the traditional one. The calculation of geographic median consumes 30% more time than the calculation of mean on 160 GB data since it is more complex and requires more iterations to approximate the optimal point.



(a) knn



(b) distribution

Figure 7.5: Results of KNN and Distribution.

Figure 7.5(a) measures the performance of computing KNN, where k value is 10, when increasing the input data size from 20 GB to 160 GB. As it shows, the time does not grow linearly, but relatively slowly. Figure 7.5(b) shows the performance trend of geometric median distribution, which is similar to KNN. We only spend 1 minute to finish KNN and geometric median distribution on 160 GB dataset, which demonstrates that the Dart system is quite scalable.

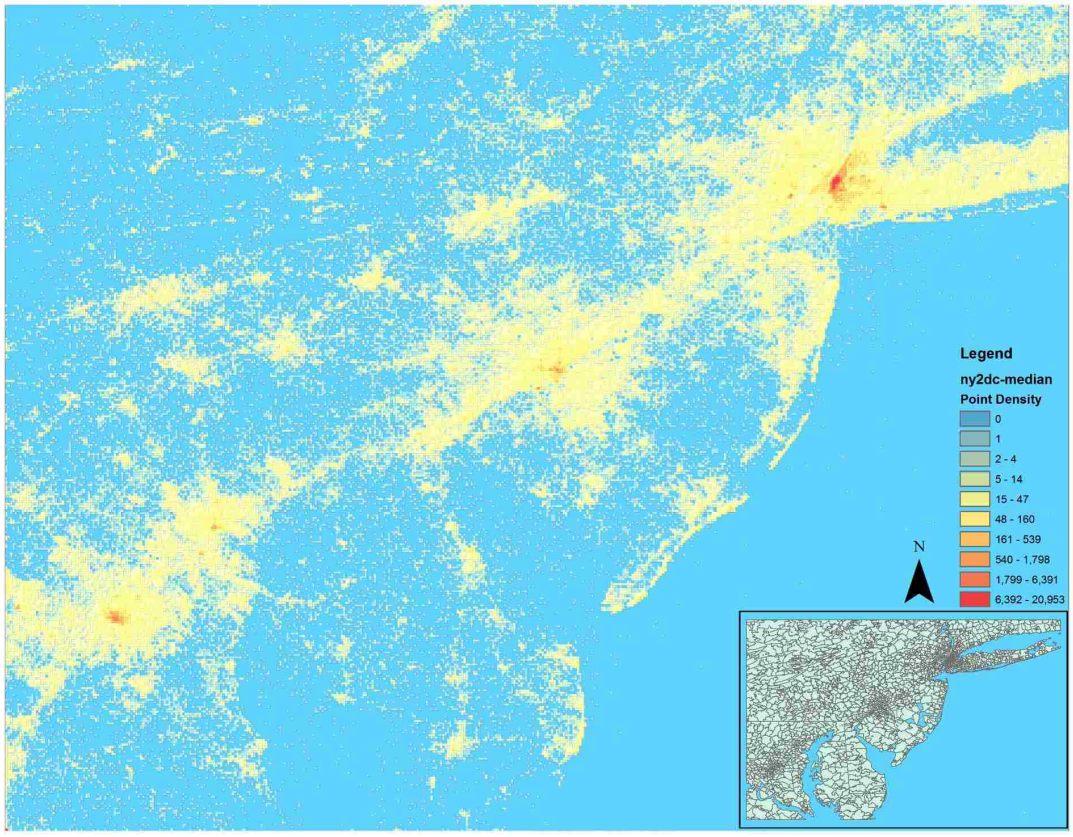


Figure 7.6: Results of Distribution

Figure 7.6 shows the median centers of all mobile Twitter users discovered from the real tweet dataset, which covers a geographic area from Washington, DC to New York City. The distribution pattern of Twitter users' daily activity locations reveals that Twitter users are more likely to live in

urban areas.

The main contribution of this study is in the development of a system for rapid spatial data analysis. The crafting of this system lays a solid foundation for future research that will look into the spatio-temporal patterns as well as the socio-economic characteristics of a massive population, which are crucial inputs for effective urban planning or transportation management.

CHAPTER 8: CONCLUSION AND FUTURE WORK

In this section, we summarize conclusion remarks of the proposed designs.

First of all, we introduce an asynchronous multi-pipeline file transfer protocol with a revised fault tolerance mechanism instead of the HDFS's default stop-and-wait single-pipeline protocol. We employ global and local optimization techniques to sort datanodes in pipelines based on the historical data transfer speed. We conduct a series of experiments on Amazon's EC2 by varying the instance type, number of instances, and network bandwidth. Our experiments reveal significant improvement by 27-245% compared with HDFS.

Secondly, we present an optimized Hadoop system to improve the performance of short jobs in two modes: D+ mode and U+ mode. In D+ mode, we design a new scheduler to schedule Map tasks based on the resource distribution situation and data locality. Instead of waiting for heartbeats reported from NMs to decide how to schedule tasks, our scheduler can allocate resources immediately. Our algorithm not only avoids allocation imbalance problem for short jobs, but also reduces the communication cost. For the U+ mode, rather than executing Map tasks sequentially, we employ multi-threading to run Map tasks in parallel. In addition, we cache the intermediate data into memory instead of writing them into disk and reading them later when the intermediate data are small. Moreover, we implement a new job submitting framework and speculative execution system to reduce the setup cost for short jobs. Our experiments show that our system can obtain significant performance improvement by 11% to 88% compared with the original Hadoop for short jobs.

Thirdly, we design an end-to-end performance model for Spark, and use 3-level sampling model to sample the test application, *i.e.*, input data level, task level, and element level. We also introduce a new white box performance model to evaluate the performance based on "Law of Diminishing

Marginal Utility". Initial results show that our optimization can gain 19.6% performance improvement compared to the naive configuration, even by tuning only 3 parameters.

Finally, we design an asynchronous parallel SGD algorithm with iterative local search, called FTSGD, by reusing data partition multiple times within a single global iteration, and prove the convergence property. We also design a novel geographic information system named Dart for spatial data analysis and management. Using a hybrid table schema to store spatial data in HBase, Dart can omit the Reduce process for operations like calculating the mean center and the median center. It also employs pre-splitting and hash techniques to avoid data imbalance and hot region problems. As demonstrated in our experiments, Dart achieves a significant improvement in computing performance in contrast to the performance of traditional GIS.

Due to its speed and ease of use, Spark has become a popular tool amongst data scientists to analyze data in various sizes. In the future, we will extend our aforementioned Hadoop solutions to Spark to speedup its performance based on the improved storage and resource management layers. In addition, we will enhance our what-if engine and optimizer for Spark to give an optimal configuration for every type of application by timely and cost-effective analytics.

APPENDIX A: COPYRIGHT PERMISSION

INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING

To ensure uniformity of treatment among all contributors, other forms may not be substituted for this form, nor may any wording of the form be changed. This form is intended for original material submitted to the IEEE and must accompany any such material in order to be published by the IEEE.Please read the form carefully and keep a copy for your files.

TITLE OF PAPER/ARTICLE/REPORT, INCLUDING ALL CONTENT IN ANY FORM, FORMAT, OR MEDIA (hereinafter, "The Work"):**SMARTH: Enabling Multi-pipeline Data Transfer in HDFS**

COMPLETE LIST OF AUTHORS:**Hong Zhang**

IEEE PUBLICATION TITLE (Journal, Magazine, Conference, Book):**2014 43nd International Conference on Parallel Processing**

COPYRIGHT TRANSFER

1. The undersigned hereby assigns to The Institute of Electrical and Electronics Engineers,Incorporated (the "IEEE") all rights under copyright that may exist in and to: (a) the above Work,including any revised or expanded derivative works submitted to the IEEE by the undersigned based on the Work; and (b) any associated written or multimedia components or other enhancements accompanying the Work.

Author: **Hong Zhang**

Date: **03-07-2014**

APPENDIX B: COPYRIGHT PERMISSION

INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM

To ensure uniformity of treatment among all contributors, other forms may not be substituted for this form, nor may any wording of the form be changed. This form is intended for original material submitted to the IEEE and must accompany any such material in order to be published by the IEEE.Please read the form carefully and keep a copy for your files.

MRapid: An Efficient Short Job Optimizer on Hadoop

Hong Zhang, Hai Huang, Liqiang Wang

2017 IEEE International Parallel and Distributed Processing Symposium

COPYRIGHT TRANSFER

1. The undersigned hereby assigns to The Institute of Electrical and Electronics Engineers,Incorporated (the "IEEE") all rights under copyright that may exist in and to: (a) the above Work,including any revised or expanded derivative works submitted to the IEEE by the undersigned based on the Work; and (b) any associated written or multimedia components or other enhancements accompanying the Work.

Author: **Hong Zhang**

Date: **27-02-2017**

APPENDIX C: COPYRIGHT PERMISSION

INTERNATIONAL CONFERENCE ON CLOUD ENGINEERING

To ensure uniformity of treatment among all contributors, other forms may not be substituted for this form, nor may any wording of the form be changed. This form is intended for original material submitted to the IEEE and must accompany any such material in order to be published by the IEEE.Please read the form carefully and keep a copy for your files.

Tuning Performance of Spark Programs

Hong Zhang, Zixia Liu, Liqiang Wang

2018 IEEE International Conference on Cloud Engineering

COPYRIGHT TRANSFER

1. The undersigned hereby assigns to The Institute of Electrical and Electronics Engineers,Incorporated (the "IEEE") all rights under copyright that may exist in and to: (a) the above Work,including any revised or expanded derivative works submitted to the IEEE by the undersigned based on the Work; and (b) any associated written or multimedia components or other enhancements accompanying the Work.

Author: **Hong Zhang**

Date: **13-02-2018**

APPENDIX D: COPYRIGHT PERMISSION

INTERNATIONAL CONFERENCE ON CLOUD COMPUTING

To ensure uniformity of treatment among all contributors, other forms may not be substituted for this form, nor may any wording of the form be changed. This form is intended for original material submitted to the IEEE and must accompany any such material in order to be published by the IEEE.Please read the form carefully and keep a copy for your files.

TITLE OF PAPER/ARTICLE/REPORT, INCLUDING ALL CONTENT IN ANY FORM, FORMAT, OR MEDIA (hereinafter, "The Work"):**Dart: A Geographic Information System on Hadoop**

COMPLETE LIST OF AUTHORS:**Hong Zhang**

IEEE PUBLICATION TITLE (Journal, Magazine, Conference, Book):**2015 IEEE 8th International Conference on Cloud Computing**

COPYRIGHT TRANSFER

1. The undersigned hereby assigns to The Institute of Electrical and Electronics Engineers,Incorporated (the "IEEE") all rights under copyright that may exist in and to: (a) the above Work,including any revised or expanded derivative works submitted to the IEEE by the undersigned based on the Work; and (b) any associated written or multimedia components or other enhancements accompanying the Work.

Author: **Hong Zhang**

Date: **05-05-2015**

LIST OF REFERENCES

[1] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component sys*, 30:19, 2002.

[2] H. Zhang, L. Wang, and H. Huang. Smarth: Enabling multi-pipeline data transfer in hdfs. In *ICPP*, 2014.

[3] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI 2004*, pages 1–13, 2004.

[4] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *8th USENIX conference on Operating systems design and implementation*, pages 1–14, 2008.

[5] K. Elmeleegy. Piranha: optimizing short jobs in hadoop. In *VLDB Endowment*, pages 985–996, 2013.

[6] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *ACM SIGMOD*, pages 13–24, 2013.

[7] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing a sql implementation on the mapreduce framework. In *VLDB Endowment*, pages 1318–1327, 2011.

[8] H. Zhang, H. Huang, and L. Wang. Mrapid: An efficient short job optimizer on hadoop. In *IPDPS*, 2017.

[9] https://databricks.com/spark/about. Databricks Spark website.

[10] https://spark.apache.org/. Apache Spark website.

[11] Hong Zhang, Zixia Liu, and Liqiang Wang. Tuning performance of spark programs. In *Cloud Engineering (IC2E), 2018 IEEE International Conference on*, pages 282–285. IEEE, 2018.

[12] H. Zhang, Z. Sun, Z. Liu, C. Xu, and L. Wang. Dart: A geographic information system on hadoop. In *IEEE CLOUD*, pages 1–8. IEEE, 2015.

[13] P. Russom. *Big Data Analytics*. TDWI Research, 2011.

[14] D.W. Wong and J. Lee. *Statistical Analysis and Modeling of Geographic Information*. John Wiley & Sons, New York, 2005.

[15] C. Xu, D.W. Wong, and C. Yang. Evaluating the "geographical awareness" of individuals: an exploratory analysis of twitter data. In *CaGIS 40(2)*, pages 103–115, 2013.

[16] Amazon EC2 website. http://aws.amazon.com/ec2/.

[17] Weijia Xu, Wei Luo, and Nicholas Woodward. Analysis and optimization of data import with hadoop. In *IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1–9, 2012.

[18] Jeffrey Shafer, Scott Rixner, and Alan L. Cox. The hadoop distributed filesystem balancing portability and performance. In *IEEE International Symposium on Performance Analysis of Systems & Software*, pages 1–12, 2010.

[19] Mohamed Y. Eltabakh, Yuanyuan Tian, Fatma Ozcan, Rainer Gemulla, Aljoscha Krettek, and John McPherson. Cohadoop: flexible data placement and its exploitation in hadoop. In *the VLDB Endowment*, pages 575–585, 2011.

[20] Kun Lu, Dong Dai, and Mingming Sun. Hdfs+: Concurrent writes improvements for hdfs. In *the 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 1–2, 2013.

[21] N.S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High performance rdma-based design of hdfs over infiniband. In *High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2012.

[22] Adam Yee and Jeffrey Shafer. Hfaa: a generic socket api for hadoop file systems. In *the 2nd Workshop on Architectures and Systems for Big Data*, pages 15–20, 2011.

[23] Yandong Wang, Xinyu Que, Weikuan Yu, Dror Goldenberg, and Dhiraj Sehgal. Hadoop acceleration through network levitated merge. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2011.

[24] Y. Yao, J. Tai, B. Sheng, and N. Mi. Lsps: A job size-based scheduler for efficient task assignments in hadoop. In *IEEE Transactions on Cloud Computing*, pages 411 – 424, 2014.

[25] J. Yan, X. Yang, R. Gu, C. Yuan, and Y. Huang. Performance optimization for short mapreduce job execution in hadoop. In *CGC*, pages 1–7, 2012.

[26] M.Hammoud and M. F. Sakr. Locality-aware reduce task scheduling for mapreduce. In *CloudCom*, pages 570–576, 2011.

[27] X. Zhang, Z. Zhong, S. Feng, B. Tul, and J. Fan. Improving data locality of mapreduce by scheduling in homogeneous computing environments. In *Int. Symp. on Parallel and Distributed Processing with Applications*, pages 120–126, 2011.

[28] S. Ibrahim, H. Jin, L. Lu, B. He, G. Antoniu, and S. Wu. Maestro: Replica-aware map scheduling for mapreduce. In *CCGRID*, pages 435–442, 2012.

[29] L. George. *HBase: The Definitive Guide*. O'Reilly Media, 2011.

[30] Y. Zhang. Hj-hadoop: An optimized mapreduce runtime for multi-core systems. In *SPLASH*, pages 111–112, 2013.

[31] Apache Spark website. http://Spark.apache.org/.

[32] S. Ryza, U. Laserson, S. Owen, and J. Wills. *Advanced Analytics with Spark: Patterns for Learning from Data at Scale*. O'Reilly Media, 2015.

[33] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *ACM SIGCOMM*, pages 407–420, 2015.

[34] J. Urbani, A. Margara, C. Jacobs, S. Voulgaris, and H. Bal. Ajira: A lightweight distributed middleware for mapreduce and stream processing. In *ICDCS*, pages 545–554, 2014.

[35] Spark management website. https://0x0fff.com/spark-memory-management/.

[36] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, 2011.

[37] K. Wang and M. Khan. Performance prediction for apache spark platform. In *HPCC*. IEEE, 2015.

[38] L. Xu, M. Li, L. Zhang, A. Butt, Y. Wang, and Z. Hu. Memtune: Dynamic memory management for in-memory data analytic platforms. In *IPDPS*. IEEE, 2016.

[39] A. Paul, W. Zhuang, L. Xu, M. Li, M. Rafique, and A. Butt. Chopper: Optimizing data partitioning for in-memory data analytics frameworks. In *CLUSTER*, pages 110–119. IEEE, 2016.

[40] R. Agrawal, A. Somani, and Y. Xu. Storage and querying of e-commerce data. In *VLDB Endowment*, pages 149–158, 2001.

[41] A. Eldawy and M. Mokbel. SpatialHadoop: towards flexible and scalable spatial processing using mapreduce. In *the 2014 SIGMOD PhD symposium*, pages 46–50, New York, NY, USA, 2014.

[42] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz. Hadoop-GIS: A high performance spatial data warehousing system over mapreduce. In *VLDB Endowment*, pages 1009–1020, 2013.

[43] S. Nishimura, S. Das, D. Agrawal, and A. Abbadi. MD-HBase: A scalable multi-dimensional data infrastructure for location aware services. In *MDM*, pages 7 – 16, 2011.

[44] A. Eldawy, Y. Li, M. Mokbel, and R. Janardan. CG-Hadoop: Computational geometry in mapreduce. In *SIGSPATIAL*, pages 294–303, 2013.

[45] S. Zhang, J. Han, Z. Liu, K. Wang, and S. Feng. Spatial queries evaluation with mapreduce. In *GCC*, pages 287 – 292, 2009.

[46] W. Lu, Y. Shen, S. Chen, and B. Ooi. Efficient processing of k nearest neighbor joins using MapReduce. In *VLDB Endowment*, pages 1016–1027, 2012.

[47] Y. Liu, K. Wu, S. Wang, Y. Zhao, and Q. Huang. A mapreduce approach to Gi*(d) spatial statistic. In *HPDGIS*, pages 11–18, 2010.

[48] K. Al-Naami, S. Seker, and L. Khan. Gisqf: An efficient spatial query processing system. In *CLOUD*, pages 681 – 688, 2014.

[49] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2012.

[50] Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D. Bowers, and Michael M. Swift. More for your money: exploiting performance heterogeneity in public clouds. In *the Third ACM Symposium on Cloud Computing*, pages 1–14, 2012.

[51] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *NSDI*, pages 323–336, 2011.

[52] Apache Spring website. http://projects.spring.io/spring-hadoop/.

[53] Asm website. http://asm.ow2.org/.

[54] Microsoft Azure website. https://azure.microsoft.com.

[55] Ji Liu, Stephen J Wright, Christopher Ré, Victor Bittorf, and Srikrishna Sridhar. An asynchronous parallel stochastic coordinate descent algorithm. In *Machine Learning Research 16*, 1988.

[56] S. Zhao and W. Li. Fast asynchronous parallel stochastic gradient descent: A lock-free approach with convergence guarantee. In *AAAI-16*, 2016.

[57] F. Niu, B. Recht, C. Re, and S. Wright. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, 2011.

[58] M. A. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized stochastic gradient descent. In *NIPS*, 2010.

[59] S. Zhang, C. Zhang, Z. You, R. Zheng, and B. Xu. Asynchronous stochastic gradient descent for dnn training. In *ICASSP*, 2013.

[60] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[61] T. Paine, H. Jin, J. Yang, Z. Lin, and T. Huang. Gpu asynchronous stochastic gradient descent to speed up neural network training. In *CVPR*, 2013.

[62] R. A. Jacobs. Increased rates of convergence through learning rate adaptation. In *Neural Networks:Volume 1, Issue 4*, 1988.

[63] D. J. Swanston, J. M. Bishop, and R. J. Mitchell. Simple adaptive momentum: new algorithm for training multilayer perceptrons. In *Electronics Letters*, 1994.

[64] N. M. Nawi, R. S. Ransing, and M. R. Ransing. An improved conjugate gradient based learning algorithm for back propagation neural networks. In *International Journal of Computational Intelligence 4*, 2008.

[65] P. Guo, H. Huang, Q. Chen, L. Wang, E. Lee, and P. Chen. A model-driven partitioning and auto-tuning integrated framework for sparse matrix-vector multiplication on gpus. In *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*. ACM, 2011.

[66] H. Huang, J. Dennis, L. Wang, and P. Chen. A scalable parallel lsqr algorithm for solving large-scale linear system for tomographic problems: a case study in seismic tomography. *Procedia Computer Science*, 18, 2013.

[67] H. Zhang, C. Hsieh, and V. Akella. Hogwild++: A new mechanism for decentralized asynchronous stochastic gradient descent. In *ICDM*, 2016.

[68] C. Lin, C. Tsai, C. Lee, and C. Lin. Large-scale logistic regression and linear support vector machines using spark. In *IEEE Big Data*, 2014.

[69] Z. Liu, H. Zhang, , and L. Wang. Hierarchical spark: A multi-cluster big data computing framework. In *IEEE Cloud*, 2017.

[70] Wikipedia for kolmogorov-smirnov test. https://en.wikipedia.org/wiki/Kolmogorov-Smirnov_test.

[71] Wikipedia website for cosine similarity. https://en.wikipedia.org/wiki/Cosine_similarity.

[72] Website for calculating mean, midpoint, and center of minimum distance. http://www.geomidpoint.com/.