IMPROVEMENT OF DATA INTENSIVE APPLICATIONS RUNNING ON CLOUD COMPUTING CLUSTERS

by

IBRAHIM ADEL IBRAHIM
B.S. AlMustansiriyah University, Baghdad, 2002
M.Sc. University Of Technology, Baghdad, 2006

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

Fall Term
2019

Major Professor: Mostafa Bassiouni

ABSTRACT

MapReduce, designed by Google, is widely used as the most popular distributed programming model in cloud environments. Hadoop, an open-source implementation of MapReduce, is a data management framework on large cluster of commodity machines to handle data-intensive applications. Many famous enterprises including Facebook, Twitter, and Adobe have been using Hadoop for their data-intensive processing needs. Task stragglers in MapReduce jobs dramatically impede job execution on massive datasets in cloud computing systems. This impedance is due to the uneven distribution of input data and computation load among cluster nodes, heterogeneous data nodes, data skew in reduce phase, resource contention situations, and network configurations. All these reasons may cause delay failure and the violation of job completion time. One of the key issues that can significantly affect the performance of cloud computing is the computation load balancing among cluster nodes. Replica placement in Hadoop distributed file system plays a significant role in data availability and the balanced utilization of clusters. In the current replica placement policy (RPP) of Hadoop distributed file system (HDFS), the replicas of data blocks cannot be evenly distributed across cluster's nodes. The current HDFS must rely on a load balancing utility for balancing the distribution of replicas, which results in extra overhead for time and resources. This dissertation addresses data load balancing problem and presents an innovative replica placement policy for HDFS. It can perfectly balance the data load among cluster's nodes. The heterogeneity of cluster nodes exacerbates the issue of computational load balancing; therefore, another replica placement algorithm has been proposed in this dissertation for heterogeneous cluster environments. The timing of identifying the straggler map task is very important for straggler mitigation in data-intensive cloud computing. To mitigate the straggler map task, Present progress and Feedback based Speculative Execution (PFSE) algorithm has been proposed in this dissertation. PFSE is a new straggler identification scheme to identify the straggler map tasks based on the feedback information received from completed tasks beside the progress of the

current running task. Straggler reduce task aggravates the violation of MapReduce job completion time. Straggler reduce task is typically the result of bad data partitioning during the reduce phase. The Hash partitioner employed by Hadoop may cause intermediate data skew, which results in straggler reduce task. In this dissertation a new partitioning scheme, named Balanced Data Clusters Partitioner (BDCP), is proposed to mitigate straggler reduce tasks. BDCP is based on sampling of input data and feedback information about the current processing task. BDCP can assist in straggler mitigation during the reduce phase and minimize the job completion time in MapReduce jobs. The results of extensive experiments corroborate that the algorithms and policies proposed in this dissertation can improve the performance of data-intensive applications running on cloud platforms.

ACKNOWLEDGMENTS

I owe a debt of gratitude to my major advisor Prof. Mostafa Bassiouni for his efforts, support, and encouragement throughout my doctoral study. He was always keeping me motivated during my research. It's been a great honor for me to conduct my doctoral research under his supervision. To him, I am and will always be deeply grateful.

I am very thankful to Dr. Qun Zhou, Dr. Mingjie Lin, Dr Rickard Ewetz, and Dr. Ivan Garibay for serving on my dissertation committee.

Thank you for my brothers, Salam, Wisam, and Musaab, and my sisters Athraa, and Zahraa. They have provided me with their love, and encouragement.

I would love to thank my soul mate, my friend, and my wife, Sarah Aljumaily, who has never quit on believing on me and supported me in my life and in the journey of PhD research. Thank you Sarah for supporting and encouraging me.

I would like to thank my first teacher, the great women, my mother. She has been waiting for this moment and she has been keeping me in her prayer every day.

One final word of gratitude goes to the great man who has always waited for this moment, a person whom I always dreamt of writing these words to, a person who saw his dream in me, and he is suffering now from Lung Cancer but he promised me that he will be stronger than the cancer and he is going to be presented in my graduation; my father. I am sorry I could not make it earlier, but i am sure you are stronger than the cancer and you will be presented in my graduation. Thank you is not enough!

To Fahad and Ellen

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

The advances in Information Technology and the storage demands of cloud computing have been growing rapidly in the last few years. Storage systems are under intense pressure due to the explosive growth of data amount generated through many distributed applications, such as search engines, social networking sites, grid computing applications, data mining applications, etc. An efficient file system is required to store internet generated large data and effective handling of huge files. Cloud computing is an emerging technology that attracts data service providers offering tremendous opportunities for online distribution of services. It offers computing as a utility, sharing resources of expandable data centers [2]. End users can benefit from the convenience of accessing data and services globally, centrally managed backups, and high computational .

1.1 Mapreduce

The rapid growth of information and data in the age of data explosion in industry and research poses tremendous opportunities, as well as tremendous computational challenges. To manage the immense volumes of data, users have needed new systems to scale out computations to multiple nodes. Modern cloud datacenters are composed of thousands of servers to support the increasing demand on cloud computing. Due to the large scale of the data-intensive jobs, the only feasible way to solve them while fulfilling Quality of Service (QoS) requests is to partition them into small tasks which can be processed in parallel across many computing nodes [4]. MapReduce, designed by Google, has been widely used as the most popular distributed programming model for parallel processing of massive datasets (usually greater than 1 TB) in cloud environments. It divides a large computation into small tasks and assigns them to multiple computational cluster of nodes running in parallel. MapReduce is unique in reliability, and scalability to large clusters of inexpensive commodity computers. It automatically partitions a job into multiple tasks and transparently handles

parallel tasks execution and complexity of fault tolerance from the programmer in a distributed manner [69].

MapReduce consists of two main phases: map, and reduce. Tasks are distributed to cluster of processing nodes during Map and Reduce phases. During Map phase, chunks of the huge data sets are processed concurrently on individual computers in the cluster. The Map function is applied to an individual input record to compute a set of intermediate key/value pairs. For each key, Reduce works on the list of all values with this key [35]. Reduce phase has 3 steps: shuffle, sort, and reduce. Shuffle starts when the data is collected by the reducer from each mapper. Shuffle may start when mappers have generated enough amount of data. On the other hand, sort and reduce can start once all the mappers have finished map phase and the resulted intermediate data have been shuffled to the reducers. Reduce phase combines the intermediate data from Map phase and derives the final output.

1.2 Hadoop

Hadoop, an open source implementation of MapReduce, is a data management framework on large cluster of commodity machines to handle large-scale data intensive applications. Hadoop is a solution that provides reliability, scalability, and manageability of big data [64]. Hadoop divides a large computation into small tasks and assigns them to multiple computational cluster nodes running in parallel. It manages the parallel processes on a large file, where the file is divided into many chunks distributed on clusters of inexpensive commodity computers, automatically handles failures, and hides the complexity of fault tolerance from the programmer. It scales easily to large clusters of inexpensive commodity computers [68]. To build such a cloud system, an increasing number of companies and academic institutions have started to rely on the Hadoop Distributed File System HDFS. The next generation of Hadoop, namely Hadoop YARN, is accommodated to various programming frameworks and capable of handling many kinds of workload such as

interactive analysis, and stream processing.

1.3 Hadoop Distributed File System HDFS

In order to meet the ever-growing data storage demands from users, The storage system for cloud computing consolidates large numbers of distributed commodity computers into a single storage pool to provide online services for data storage with immense capacity and high quality of service in an unreliable and dynamic network environment at low cost [75]. To build such a cloud storage system, an increasing number of companies and academic institutions have started to rely on the Hadoop Distributed File System (HDFS) [57]. Many cloud vendors have given attractive storage service offerings that provide a giant cloud-based storage space for users, such as Amazon, Dropbox, Google Drive, and Microsoft's OneDrive [23, 25, 28]

HDFS has been widely used and become a common storage appliance for cloud computing. It is the storage part of Hadoop framework, it is a distributed, scalable, and portable file system designed to run on low-cost hardware. Although it has many similarities with other existing distributed file systems, the differences from other distributed file systems are significant. HDFS is especially designed to be highly fault-tolerant, and to provide high throughput access to application data and is suitable for applications that have large data sets.

The idea of blocks is exist in many file systems. Block is the smallest unit which is loaded into memory in one operation. However HDFS is dealing with very large files, a block in HDFS is the smallest replication unit. All blocks in a file except the last block are the same size, the default block size is 64MB. In HDFS each data file is stored as a sequence of blocks. A file is split into blocks during the write operation and distributed across cluster nodes. Also depending on the client application, a block is, usually, the data unit on which an application copy operates on. Blocks of a file are replicated for reading performance and fault tolerance. More replicas could enhance the availability of files on cloud systems and provide users a better chance to retrieve their data when

serious disasters occur [73]. HDFS uses an uniform triplicating policy (i.e. three replicas for each data block) to improve data locality and ensure data availability and fault tolerance in the event of hardware failure [57]. The placement policy of replicas is critical to HDFS performance and reliability. This policy could also achieve load balancing by distributing work across the replicas. For the common case, the triplication policy in HDFS works well in term of high reliability and high performance. The consequence of the current replica placement policy is that the hadoop cluster gets unbalanced in terms of both storage load and the data processing load. However, the more data a node contains, the higher probability the node could be selected to serve the data. An unbalanced cluster puts a greater strain on the highly utilized DataNodes, and results in straggler tasks in MapReduce jobs [68].

1.4 Straggler task

In MapReduce, after a job is submitted, the input file is divided into multiple map tasks, and then both map and reduce tasks are assigned to multiple data nodes. Slow task or task with more data becomes straggler. The straggler task degrades the performance of MapReduce applications because it delays the final results. In cloud computing platform, stragglers are very common problem especially in the case of data-intensive computing jobs because of two major reasons. First, cloud data centers use commonly commodity hardware instead of expensive, powerful, and highly reliable hardware. As a result, the probability of part failure is high for large clusters. Part failure may just degrade the node performance Instead of causing complete node failure, which would generate stragglers. For example, a node, with a faulty hard drive may experience frequent read errors that are correctable, can still work but with very slow disk read speed.

Second, virtualization technology, such as the Amazon Elastic Compute Cloud (EC2) [5], is widely used by service providers of utility computing to provide an abstraction of the underlying hardware. Even though virtualization technology isolates the CPU and memory usage, both

disk and network bandwidth are still shared among virtual machines residing on the same physical host, which can cause notable heterogeneity in virtual machine performance as the resource contention, the growth of virtualization have further aggravated the heterogeneity, as a result, all virtual machines on overloaded physical host become stragglers. Thus, dealing with the problem of stragglers becomes critical and significant because stragglers can seriously impact the completion time of parallel processing jobs.

Hadoop employs a mechanism called speculative execution to deal with the straggler issue, it runs a speculative copy of a straggler's task on another normal node, the default speculative algorithm in Hadoop aggressively starts many backup tasks. Although speculative execution can dramatically reduce the job completion time because, in most cases, the speculative copy completes much earlier than the original task running on the straggler, but some of the straggler finish before the backup tasks, in this case the backup becomes ineffective and results in insufficient resource consuming. So the Identification of the straggler is the most crucial part of speculative execution, because accuracy in identifying the straggler can significantly improve job completion time and sufficiently increase the resource utilization by avoiding running backup task that will be discarded if it's original task was incorrectly identified as straggler, where the results of the earlier completed task is taken and the other task is ignored. So stragglers must be detected correctly and early enough to get better efficiency of the speculative execution mechanism.

1.5 Dissertation organization

The dissertation is organized as follows. Chapter 2 describes the existing research related to our work. Chapter 3 presents a balancing policy for data placement in cloud storage systems. An innovative replica placement policy for HDFS in heterogeneous cluster environments is presented in Chapter 4. Furthermore, Chapter 5 presents a new straggler identification scheme to make Hadoop more efficient in cloud environments. Subsequently, in Chapter 6 a new partitioning scheme, called

balanced data clusters partitioner (BDCP), is proposed to handle straggler reduce tasks in MapReduce jobs. Finally, Chapter 7 discusses the conclusion of the dissertation.

CHAPTER 2: LITERATURE REVIEW

In this chapter, some literature that related to this research work are reviewed

2.1 Replicas distribution policy in Hadoop distributed file system

The drawback of the existing block placement policy of hadoop is that it does not distribute replicas of blocks fairly and evenly across cluster's nodes [8]. Large amount of research work has been conducted on the HDFS RPP due to its importance regarding improving the performance of HDFS. Shabeera et al. proposed a RPP for Hadoop that depends on the available bandwidth between the HDFS client and cluster nodes in [56], the bandwidth can be measured and compared periodically, and the node that has the maximum bandwidth is selected to place the replica on it in order to reduce the time of data transfer. Khan et al. [38] presents an algorithm that finds the optimal number of codeword symbols needed for recovery for any XOR-based erasure code and produces recovery schedules that use a minimum amount of data. It improves I/O performance in practice for the large block sizes used in cloud file systems, such as HDFS.

Zhang et al. present Aurora [78], a dynamic block placement mechanism, which implements several local search algorithms in HDFS. They propose several constant-factor local search approximation algorithms, and present a dynamic replica distribution mechanism that implements the algorithms in HDFS. The results of Experiments show Aurora can solve the dynamic block replication problem and remarkably reduce the uneven load distribution, also it meets all the rank level reliability requirements of HDFS. Long et al.[52] proposed Multi-objective Optimized Replication Management (MORM). It is an improved artificial immune algorithm used for optimizing file unavailability, service time, load variance, energy consumption and access latency by managing the replication factor and replica placement in VMs. Lin et al. [48] proposed a strategy where the NameNode chooses DataNodes according to the load status. In this strategy a new node named

BalanceNode is proposed. It is used to match heavy-loaded and light-loaded DataNodes, so that the light-loaded DataNodes can share the load with heavy-loaded ones. In our previous work [20] we address the load balancing issue from the perspective of balancing replicas assignment across all cluster nodes of Hadoop, and propose a new placement policy that split the nodes in to three different sections and run an algorithm to distribute the replicas on the three sections across all cluster nodes. Zhou et al. [79] proposed BigRoots algorithm to identify the root causes of the stragglers. BigRoots incorporates both framework and system features for root-cause analysis of stragglers in the big data system. The results of extensive experiments corroborate that BigRoots is effective for identifying the root causes of stragglers and providing useful indicators for performance optimization.

Nonava et al. [53] proposed a policy based on the processing power of the hardware generation. They proposed placing more data blocks onto newer hardware generation nodes. In this policy, they calculate a quota based on the hardware generation of the Data Nodes during placing blocks. Blocks will be placed on the nodes which have the highest quota. Lee et al. [42] proposed a dynamic block placement algorithm. this algorithm places blocks on the DataNodes based on their processing capacity. in this Algorithm, NameNode creates a RatioTable, it is used to determine the ratio of data blocks that decided to be placed on each DataNode. However [42, 53] depends only on the processing capacity of the DataNodes for placing data blocks. it does not consider the storage capacity of the DataNodes. The storage capacity gets unbalanced when the higher processing capacity DataNodes get greater number of data blocks. therefore, DataNodes with high processing capacity will be over utilized while the DataNodes with low processing capacity remain under utilized in terms of storage. Both algorthims result in an imbalanced storage load on the DataNodes.

Dai et al. in [18] address the load balancing issue from the perspective of task assignment of Hadoop. It propose an improved task assignment scheme that strives to balance the map task processing load of MapReduce jobs across all cluster nodes. Cheng et al. simulat[17] proposed

Elastic Replication Management System ERMS for HDFS. ERMS provides an active/standby storage model for HDFS to where the replication policy is elastic and adapt to data popularity in order to get better performance and disk utilization. ERMS has a real-time event processing engine to increase the replication factor only for the files with higher popularity. To distinguish real-time data types, ERMS utilizes a complex event processing engine. It dynamically increases extra replicas for hot data, while it cleans up these extra replicas when the data cool down. Moreover, it uses erasure codes for cold data. The experiments results indicate that ERMS reduces storage overhead and improves the performance and reliability of HDFS.and . The experiments results show that ERMS effectively improves the reliability and performance of HDFS and reduces the storage overhead. Eltabakh et al. in [22], propose CoHadoop, a lightweight extension of Hadoop, it addresses the performance problem of Hadoop that it cannot collocate related data on the same set of nodes. in CoHadoop the HDFS has been extended to allow applications to define and exploit customized placement strategies to improve the performance of the system. The results of experiments on CO-Hadoop show that only when applications need to process data from multiple files, CoHadoop can remarkably outperform Hadoop. Finally, the recent research work in [24, 45, 50, 54, 77] helped with the formation of our research idea.

2.2 MapReduce tasks and speculative execution

Due to its importance to data-intensive computing, the subject of straggler identification and tolerance has received considerable amount of research attention. The mechanism of speculative execution is used in MapReduce to address the straggler problem, which performs backup execution of the remaining running tasks when the parallel processing is close to completion. Numerous speculation based techniques have been developed for straggler mitigation providing enhancement within different operational scenarios. SkewTune [41] re-partitions the data of stragglers to move it to idle slotes resulted after completing the processing of short tasks. However, moving re-

partitioned data to idle nodes may lead to nodes communication overload, which could negatively impact the computing performance.

One commonly used straggler identification scheme is the classical Standard Deviation (SD) method. It constantly monitors the performance of all processing nodes, and marks any node as straggler if its performance is significantly lower than the sample mean of the task completion times of all running nodes. Despite its wide adoption, the SD method has certain inherent limitations, which makes it not an ideal solution to the problem of straggler identification.

Dolly [6], provides a speculative execution at job level which clones small jobs with straggler tasks. Dolly launches multiple clones of every task, the output of the clone that completes first is used and the other clones are killed. But the approaches that use the duplication of the entire job increases the resource usage and I/O contention on intermediate data, in addition to that, it does not have the coordination between Hadoop and the cloud infrastructure. Dolly employs a technique called "delay assignment" to avoid contention for intermediate data. however Dolly specifically tackles stragglers in small jobs.

Xie et al. [70] propose skew data partitioning according to cluster node capabilities; slow nodes receive less work than faster nodes. This static profiling does not consider a third party loads that may begin or end in the middle of a MapReduce job, making the actual node's ability to complete work different from the profile-based prediction. Furthermore, if the nodes that predicted to receive more load fail, the application stalls for longer than when slower nodes fail. Wang et al. [67] presented an algorithm named Partial Speculative Execution (PSE), it is improved strategy to enhance the efficiency of Speculative Execution. However PSE is not designed for heterogeneous cluster's nodes while it works good in Homogeneous cluster environments. Li et al [46] presented (SECDT), it is a new speculative execution algorithm for Hadoop. SECDT was designed and implemented by calculating the completion time of a task based on C4. 5 decision tree. Tang et al. [61] proposed Speculative Execution Performance Balancing (SEPB) with a dynamic slot allocation to improve the performance of job execution.

Wu et al. [29] proposed ERUL algorithm, it calculates the prediction of system load and the remaining time of a task. ERUL improves the accuracy of the prediction by the real-time system load feedback. However this algorithm does not take into account cluster efficiency. Maximum Cost Performance (MCP) [12], a new speculative execution strategy, employs exponentially weighted moving average (EWMA) to predict the remaining time of a task. uses both progress rate and process bandwidth within a phase to identify stragglers. MCP selects the node for speculative execution using a sophisticated cost-benefit model which considers the workload on each node and the data locality. The experimental results indicate that MCP significantly improves the job performance, it can outperform Hadoop 0.21 with respect to both job completion time and cluster throughput.

There is great research interest in improving Hadoop from different perspectives. A rich set of research focused on the performance and efficiency of Hadoop cluster. After conducting a comprehensive performance study of Hadoop, Jiang et al. [34] summarized the factors that can significantly improve the Hadoop performance. Verma et al. [65] proposed cluster resource allocation approach for Hadoop. They focused on improving the cluster efficiency by minimizing resource allocations to jobs while maintaining the service level objectives. They estimated the execution time of a job based on its resource allocation and input dataset, and determined its minimum resource allocation.

There are a few recent studies focus on the improvement of speculative execution. Mantri is presented in [7], it focuses on starting speculative execution as soon as a straggler is detected during the job's execution by monitoring the performance of processing nodes in MapReduce clusters and removing stragglers based on their causes. It kills the straggler task if the speculative task is faster than the straggler. Mantri employs three major techniques: network-aware task placement, restarting tasks running on stragglers, and protecting the output of valuable tasks. Mantri estimates a task's remaining time based on the progress bandwidth. The major drawback of Mantri is there is no guarantee that the speculative copy of the straggler task will complete earlier, and it may need

to kill and restart the speculative task on multiple cluster nodes.

Wrangler [72] starts speculative tasks according to the prediction of which task would benefit from speculative execution, the prediction based on a statistical learning model that gets its facts from historical workloads. For workloads with low repetitiveness, the learning time may not be short. However, the accuracy of the prediction is affected by the performance interference in the cloud. SARS (Self-Adaptive Reduce Scheduling), a mechanism based on job context, includes the job completion time, can decide the start time points of each reduce tasks dynamically [62]. However, It focuses only on the reduce tasks. Jung et al. [36] proposed Dynamic Scheduling for Speculative Execution (DSSE) algorithm which enhances performance of the speculative execution in heterogeneous environments. DSSE prevents wasted speculative execution because it based on calculating the processing capability of each node. DSSE decreased rate of wasted speculative execution to 0%. However DSSE does not apply the speculative execution at early stages of the job execution.

Chen et al.[14] proposed Self-Adaptive MapReduce scheduling algorithm (SAMR), it uses the historical information stored on each node to create adaptable phase weights. SAMR MapReduce scheduling technique uses the historical information to find the slow nodes and launches backup tasks. According to historical information at each map stage, the time weight is adjusted and the tasks are reduced. However, SAMR does not consider the fact that different types of jobs can run on each node, therefore, different jobs have different phase weights. So SAMR cannot generate high accuracy phase weights In multi-job environment. Enhanced Self Adaptive MapReduce scheduling algorithm (ESAMR), an improved version of SAMR, was proposed by Sun et al. [59] . It groups historical information by using uses the K-mean clustering and creates a middle point of each group which contains phase weights that can be used for estimation. ESAMR calculates parameters from running tasks and compares the parameters between the running tasks and the clusters.

Lin et al. [47] proposed Self-Learning MapReduce scheduler (SLM) to improve the spec-

ulative algorithm in a multi-job cloud platform. SLM uses feedback information collected from some recently completed tasks of the same job to calculate the phase weights. However, SLM gets better accuracy of estimation with the progress of time because it needs to determine a specific number of finished tasks to use it for learning process, which affects the accuracy of the estimation process. Moreover, the estimation process is distorted by the assumption that tasks are processed at the same rate. All these approaches lack the ability to identify the performance bottleneck of the straggler tasks, they cannot guarantee that the speculative copies of tasks will perform better. Liu et al. [51] proposed LWR-SE, it is an optimized speculative execution strategy based on local data prediction in heterogeneous Hadoop environment. LWR-SE selects an appropriate node to run the backup task based on the predicted remaining time of each running tasks. LWR-SE calculates the predicted remaining time by using the collected task execution information in real time and the locally weighted regression.

Load imbalance among cluster nodes is also a major reason for the occurrence of stragglers in parallel processing. The load balance issue of Hadoop has been addresses from two different perspectives: task assignment and replica placement mechanism. The Earliest Completion Time scheme has been presented in [18] ,it is an improved task assignment scheme for Hadoop. we presented two improved replica placement policies for Hadoop, the Partition Replica Placement Policy and the Slot Replica Placement Policy, in [19, 20] respectively. Guo et al. [26] proposed, FlexSlot, a new strategy on tackling the data skew issue in Hadoop applications. Rather than mitigating skew among tasks, they try to balance the processing time of tasks. instead of balancing the distribution of data across DataNodes, tasks with expensive data records are accelerated by having more resources. FlexSlot is an effective yet simple extension to the Hadoop's slot management that provides the flexibility to alternate the number of slots in a slave node and the slot memory size online. However, despite this strategy does not need to balance the data distribution across cluster's nodes, it increases the overhead by monitoring the memory and slot size required for each task.

DynMR [60] significantly increases both performance and efficiency of Hadoop, by opportunistically schedule all tasks. It enables interleaved MapReduce execution that overlaps the map tasks with reduces tasks. Liu et al [49] proposed a new strategy called Speculation-NC, it is introduced and implemented in Hadoop-2.6. this algorthim can relatively save time and resource for WordCount sample.

2.3 Straggler reduce tasks

Many algorithms and models about reduce tasks scheduling have been proposed in recent years. Hassan et al. proposed a MRFA-Join algorithm [27], it is a new frequency adaptive algorithm based on MapReduce programming model and a randomized key redistribution approach for join processing of large-scale data sets. Data skew and load balancing problem is one of the main reasons of straggler reducers. In order to achieve balanced load, many researchers have focused on designing a new parallel programming model based on MapReduce [37, 44, 66]. LIBRA has been proposed in [13]. It is a lightweight strategy to resolve the data skew problem, it applies a sampling technique to produce an accurate estimation of the distribution of the intermediate data. It samples a part of the intermediate data during the map phase. LIBRA supports large cluster and it works for heterogeneous environments, but the partitioning does not consider the current processing load of the reducers.

Yujie et al. in [71], use a sampling MapReduce job to gather the distribution of keys' frequencies, make estimation of the overall distribution, then partition scheme is generated in advance. Two partition schemes have been proposed based on sampling results: cluster combination and cluster partition combination. The idea of cluster combination is that the biggest data cluster is assigned to the reducer with the smallest load in order to achieve the load balancing of all reducers. The cluster partition combination is used when the skew in intermediate data is very high. In this case the large cluster is divided into equal pieces, and then, every piece is assigned to a reducer.

This method breaks the rule that each partition should be processed by a single reducer. An additional reduce phase is configured to merge the results generated from multiple reducers which increase the job completion time.

Tang et al. In [63], have proposed splitting and combination algorithm for skew intermediate data blocks (SCID). The sampling is used to predict the distribution of the keys in intermediate data. In SCID, the data clusters are sorted, and for each map task the output filled into buckets. A data cluster will be split once it exceeds the residual volume of the current bucket. After filling this bucket, the remainder cluster will be started the next iteration. The main idea is that each reduce task gets its share of intermediate data from particular bucket of map task. SCID focuses on how to split and combine the output data from map tasks to the proper buckets rather than decide when the sampling should start. in addition to that, this method split a big partition to be processed by more than one reducer.

Chen et al. [10] proposed a partitioner to distribute the intermediate data in a balanced partitioning with the traditional trie, resulting in its corresponding imbalance ratio approaching one. However, their algorithm requires big amount of memory and incurs a heavy processing overhead, but they eliminated the need for big amount of memory and processing overhead required by introducing the condensed trie, which is a 3-level trie a. Then, they introduced a collapsed-condensed trie for capturing the data statistics authentically [9]. Despite their new algorithm requires less memory and processing overhead but it still requires non small amount of memory and processing overhead, and this amount depends on the sampling rate.

In our previous work [30], we proposed Progress and Feedback based Speculative Execution Algorithm (PFSE). It is a new Straggler identification scheme to identify the straggler tasks in MapReduce jobs based on the feedback information received from completed tasks, and the progress of the currently processing task. (PFSE) focuses on map phase and sort part of reduce phase only. Zaharia et al. [74] suggested Longest Approximate Time to End (LATE), a dynamic scheduling technique, was designed for heterogeneous clusters. It is modified version of specula-

tive execution. LATE allows Hadoop to speculatively execute the task that expected to be delayed. Instead of considering the progress made by a task, LATE computes the estimated remaining time to complete the task. LATE depends on HDFS for replica placement, this restriction minimizes the number of tasks that involved in the speculative execution. LATE is designed to enhance Hadoop performance in both homogeneous and heterogeneous environments. The experimental results indicate that LATE can improve the job completion time of Hadoop by a factor of two.

Ibrahim et al [33] developed an algorithm named LEEN for locality-aware and fairness-aware key partitioning in MapReduce to minimize the network bandwidth during the shuffle phase of MapReduce caused by partitioning skew. In LEEN all intermediate keys are partitioned and distributed according to their frequencies and the fairness of the expected data distribution after the shuffle phase. Chen et al [11] proposed data Locality Rather prior to data Skew (LRS) algorithm to improve the performance of MapReduce jobs. LRS is an extension of the LEEN algorithm. However LRS obtains the actual intermediate data on each map node by using Data Amount Monitor while LEEN assumes the amount of output data on each map node are equal. LRS produces data files and a metadata file. The number of data files is the number of keys. The metadata file contains a frequency table. When all map tasks are done, all metadata files will be aggregated.

Chen et al. [15] have comprehensively investigated data locality for reduce-side as well as data skew by developing Cluster Locality Partition (CLP) algorithm. It consists of three parts: Preprocess part, Data-Cluster part and Locality-Partition part. CLP uses random sampling to gather useful data information in Preprocess part, makes sure balancing load of each reducer in Data-Cluster part, and Locality-Partition part is used to partition data to right reducer. The three parts work together to improve the cluster performance. The experimental results illustrated that the CLP was better than the default partition algorithm of Hadoop in the aspects of execution time and load balancing.

However, CLP has ignored the fact that for different types of MapReduce jobs, data locality and data skew could can vary in affecting MapReduce execution time under varying in network

bandwidth. Chen et al [11] designed an extension of the CLP algorithm, it is a data Skew Rather prior to data Locality on the reduce side (SRL) algorithm, SRL algorithm is designed to increase data locality and decrease data skew on reduce side. They proposed a bandwidth-aware partitioner (BAPM), it uses the naive Bayes classifier by considering bandwidth and job type as classification attributes for proper selection SRL algorithm or LRS algorithm under various bandwidths. In our previous work [32] we address the load balancing issue from the perspective of balancing replicas assignment across all cluster nodes of hadoop, and propose a replica placement policy that run an algorithm to distribute the replicas across all cluster nodes based on their data load. In another work [31] we proposed an algorithm to address issue of load balance among heterogeneous cluster nodes. Consequently, the balancing in computational load on mappers minimizes the map phase time as well as make the reduce phase time starts earlier. Even though the distribution of intermediate data can not be predicted at the beginning of MapReduce job, balanced computational load during map phase may help in mitigating intermediate data skew if exists.

CHAPTER 3: REPLICA PLACEMENT POLICY FOR HDFS[1]

Hadoop has been widely adopted as a general-purpose platform for data-intensive computing, therefore, HDFS RPP has become popular research area to improve the data partitioning and placements. In HDFS each data file is partitioned and stored as a sequence of data blocks. Every data block is replicated to three replicas and stored on three different DataNodes to improve the processing performance data reliability. The placement of data replicas is one of the key issues that affect the performance of HDFS. Load imbalance is the major source of overhead in MapReduce jobs. Tasks with more data become stragglers and delay the overall job completion because of the uneven distribution of input data. In the current HDFS replica placement policy, the replicas of data blocks cannot be evenly distribute across cluster nodes, so the current HDFS has to rely on load balancing utility to balance replica distributions which results in extra time and resources consuming. These challenges drive the need for intelligent methods to solve the data placement problem to achieve high performance without the need for load balancing utility. In this chapter, Intelligent Data Placement Mechanism (IDPM), has been proposed to address the above challenges.

3.1 Introduction

Hadoop Distributed File System (HDFS) is a distributed storage system to stores large volumes of data reliably and provide access to the data by the applications at high bandwidth. HDFS provides high reliability and availability by storing the file as a sequence of blocks, each block is replicated, typically three copies. These replicas are distributed across multiple DataNodes. HDFS introduces a simple but highly effective policy to allocate the replicas of each data block. The default Hadoop distributed file system replica placement policy, HDFS RPP, (as of Hadoop 2.9.2)

[1]Related publication: I. A. Ibrahim, W. Dai, and M. Bassiouni. Intelligent data placement mechanism for replicasdistribution in cloud storage systems. In2016 IEEE International Conference on Smart Cloud(SmartCloud), pages 134–139. IEEE, 2016.

places one replica on one node in the local rack; another replica on a node in a remote rack; and the third on a different node in the same remote rack as shown in Fig. 3.1. This policy can reduce the inter-rack write traffic when distributing data to DataNodes because the three replicas of every data block are stored on only two racks. High fault tolerance and data availability are maintained in this policy because replicas are placed on three different DataNodes. However, The drawback of the policy is that it cannot evenly distribute replicas to cluster nodes. Data placement problem, therefore, needs to be considered to obtain an optimal placement solution to balance the data load on cluster nodes and minimize the data retrieval time [2].

3.2 Background

Even though HDFS RPP improves data reliability and processing performance, it has been clarified in our extensive experiments [32] that it cannot generate balanced replica distributions. For example: a cluster of three racks, $rack_0$ has two DataNodes, $Rack_1$ has three DataNodes, and $Rack_2$ has four data nodes; Let's assume a file is stored on Hadoop cluster nodes, the file is partitioned into 34 Blocks, every block is replicated to 3 replicas. The current replicas placement policy of HDFS generates unbalanced replicas distribution of this file as shown in Fig. 3.2. Every node gets a different number of replicas. This may result in unbalanced access load. Because the higher the data blocks stored in a DataNode the more client requests for accessing data blocks in DataNode. In a Hadoop cluster, DataNodes may not be able to handle such excessive client requests, which results in stragglar map tasks. To overcome this performance issue, HDFS provides a balancing utility to address the issue of unbalanced HDFS cluster which can seriously degrade the performance of Hadoop applications. This utility is used to analyze replica placement and re-balancing replicas distribution across the DataNodes at the cost of extra system resources and running time.

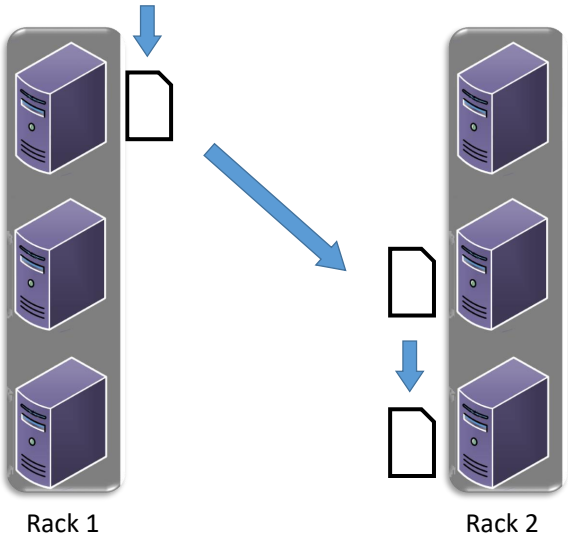

Figure 3.1: Block replication in HDFS

3.3 Balanced replicas placement policy

The unbalanced replica assignment of HDFS RPP is generated because the policy places two replicas on two different nodes belong to one rack, and the third replica on a node located in another rack. This issue increases with the increasing of the number of files distributed in the cluster. To overcome this problem, the new policy proposed in this chapter places the replicas on DataNodes with the lowest load in the cluster, at the same time, it maintains the same distribution rules of existing HDFS RPP. It is implemented by keeping pointer for each rack in the cluster to keep tracking the load on this rack. This policy makes it possible to generate an even replica distribution because the placement of replicas by using this policy is driven by the load on the DataNode instead of selecting a random racks. Since the intelligent balancing scheme is the key of the new policy, it is named the Intelligent Data Placement Mechanism Policy (IDPM).

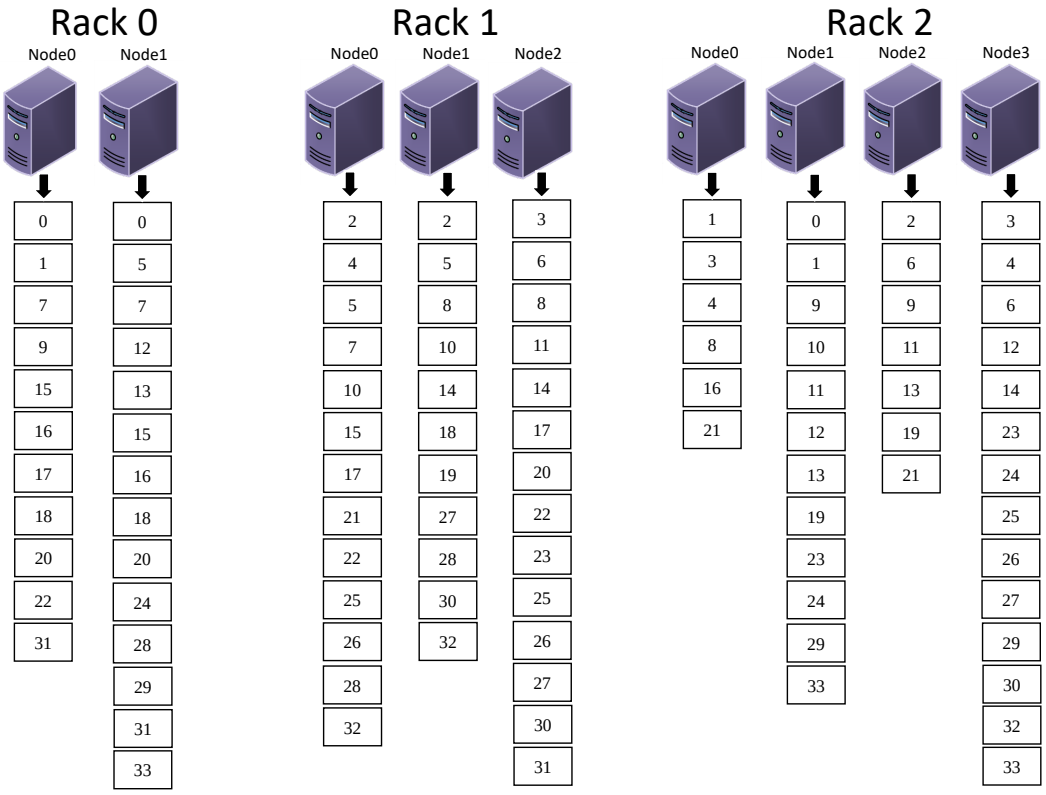

Figure 3.2: Replicas placement of HDFS RPP

In IDPM, the replica placement process consists of two phases: rack selection phase, and node selection phase. Before the placement of a replica, rack is selected then a node inside this rack is selected to place the replica on it, the rack and node selection is the core of this policy. In rack selection phase, IDPM selects two racks with the lowest load depending on a pointer called *Rack_Pointer*, it is continuously updated in every time a replica placement is achieved. The *Rack_Pointer* of a specific rack is the number of free nodes in this rack for the current iteration. the distribution process consists of many iterations as will be disscussed later in this chapter. Node selection phase starts after rack selection, in this phase a free node from the selected rack in the

current iteration is selected. In every iteration, one replica is assigned to every node. The number of iterations needed to complete the whole replicas placement can be calculated as following: Let's assume R is a collection of L racks, each of which has $r_i(i = 0,1,2,\ldots,L-1)$ available nodes on it. Let N be the total number of all available nodes. n is the total numbers ofdata blocks to be distributed, and D is the duplication factor which is assumed to be three replicas for each data block. If p is the total number of replicas to be distributed to the DataNodes in main distribution table, and I is the number of Iterations to complete the distributions, then p,N, and I can be calculated according to Eq.3.1, Eq.3.2, and Eq.3.3 respectively.

$$p = n \times D \tag{3.1}$$

$$N = \sum_{i=0}^{L-1} r_i \tag{3.2}$$

$$I = \left\lceil \frac{p}{N} \right\rceil \tag{3.3}$$

After calculating out N and I, IDPM builds the main assignment table MAT. It is two dimensional array with N columns and I rows. Every column represents a DataNode, and every row represent a distribution iteration. all the numbers of data blocks is stored in MAT. MAT consists of small tables attached together to form the main table. Each small table represents a rack, with number of columns equal to the number of nodes in that rack. The rows corresponding to the number of iterations for that rack. In each iteration of the distribution, the replicas are assigned to the whole current row before it jumps to the next iteration with the new row. The column number on the main table C which dedicated for a selected rack R, and selected node d can

be calculated according to Eq.3.4

$$C = d + \sum_{i=0}^{R-1} r_i \tag{3.4}$$

Every replica assigned to selected $rack_R$, and selected $node_d$ is stored in column C in the main assignment table. Assume there are three racks, $rack_0$ has five available nodes, $rack_1$ has three nodes, and $rack_2$ has seven nodes. As shown in Fig. 3.3 the total number of columns in the main table is N = 15. The main table consists of three sections. IDPM first assigns all five nodes on $rack_0$ to section 1, all three nodes on $rack_1$ to section 2, and then the seven nodes of $rack_2$ to section 3. The main assignment table can be formed from merging these three sections. So the main distribution table has 15 columns and I rows, where I can be found according to formula 3.3.

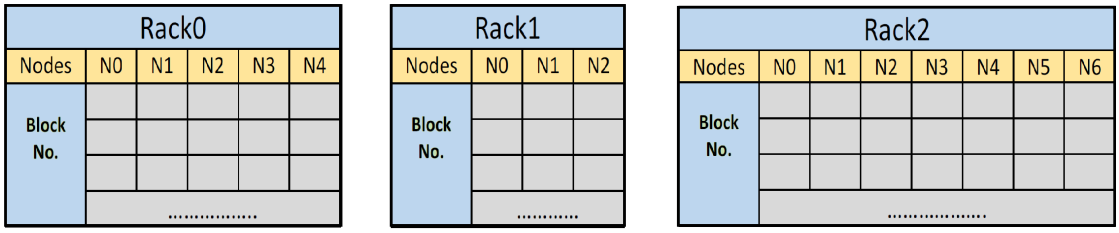

Figure 3.3: Racks tables

IDPM uses a tabular scheme to distribute replicas. the number of replica that supposed to be placed on a rack is stored in the table section corresponding to this rack. Each column in replica placement table is used to store the replicas assigned to one node. As shown in Fig. 3.4, replica's numbers are assigned into the main assignment table in block number order. Each colored section is used for the replicas of one rack, the three sections forming the main assignment table. At the beginning of distribution, two replicas of block number 0 are assigned to the section of $rack_2$.

However $rack_2$ is selected at the beginning because it is the rack that has more free nodes. Each of these two replicas are placed on different free node, $node_0$ and $node_1$, the third replica is assigned to another rack has higher free nodes, $rack_0$, on free node, $node_0$. When all the columns of the first row have been assigned replicas, the distributions start on the second row and so on.

Rack	0					1			2						
Node	0	1	2	3	4	0	1	2	0	1	2	3	4	5	6
	Main Assignment Table														
Block No.	0	1	2	2	4	3	3	4	0	0	1	1	2	3	5
	4	5	7	8	9	6	8	8	5	6	6	7	7	9	9
	10	11	13	13	15	12	12	14	10	10	11	11	12	13	14
	15	16	17	19	19	14	18	18	15	16	16	17	17	18	19
	.....								.....						

Figure 3.4: Main Assignment Table (MAT)

As shown in Fig 3.4, the replicas are assigned into the replica assignment table in block number order from replica 0 through replica $n-1$, line by line. Before assigning the third replica of current block, IDPM tags the rack in which the first two replicas of current block are residing as *avoid_rack* to discard it during the process of finding the elected rack for the third replica. After each rack election, IDPM assigns the replica to the elected node of this rack. The pseudo-code of elected rack algorithm, and elected node algorithm are shown in Alg. 1, 2 respectively. This process continues until all the p replicas have been assigned. Finally, IDPM produces an even distribution of replicas of file across all cluster nodes regardless the number of nodes in each rack in this cluster because the distribution of replicas is achieved depending on the total number of available nodes for this file. The pseudo-code of the replica distribution algorithm for all nodes is shown in Alg.3.

Algorithm 1: FUNCTION - Elected rack

```
Input: Rack_compete[0,1,..R-1]
Result: current_rack
begin
    if rack_avoide = 0 then
        rack_avoide = 0;
    end
    for i = 0 to L-1 do
        if (i = rack_avoide) then
            skip the current rack for the second third replica;
        end
        if (iteration[i] < iteration[current_rack]) then
            current_rack = i; // use the rack with the lowest iteration.
        end
        if (Rack_compete[i]>Rack_compete[current_rack])& (iteration[i] = =
          iteration[current_rack]) then
            current_rack= i; // election of the rack with more free nodes.
        end
    end
    return (current_rack)
end
```

3.4 Evaluation

In replica distribution generated by the HDFS RPP the number of replicas on one single node is a Discreet Random Variable (DRV). To test the distribution of a DRV, large scale simulation is conducted to examine the replica distribution generated by HDFS RPP, where theoretical simulation is more appropriate than actual implementation because much more distribution samples can be obtained by simulation. Two scenarios are considered, in each scenario a simulation was conducted as shown in Table 3.1. Scenario one is similar to the situations Hadoop applications encounter in practice. There are many cases in which an application has a dedicated cluster where the application can use all the nodes of racks belong to this cluster which is the case of scenario one. Assume Z is the number of replicas assigned to a single node resulted from replica distributions generated by HDFS RPP.

Algorithm 2: FUNCTION - Elected node

```
Input: current_rack
Result: current_node
1  begin
2      for i = 0 to r[current_rack] do
3          if (node_free[current_rack][i])=0 then
4              (node_free[current_rack][i])= 1;
5              current_node = i;
6              rack_compete[current_rack]= rack_compete[current_rack] -1;
7          end
8      end
9      if (rack_compete[current_rack]=0 then
10         iteration[current_rack]= iteration[current_rack] +1;
11         rack_compete[current_rack] = r[current_rack];
12         for i = 0 to r[current_rack] do
13             node_free[current_rack][j] = 0;
14         end
15     end
16     return current_node
17 end
```

Fig. 3.5 shows the probability distribution of Discreet Random Variable Z in Scenarios one, based on 15,000 simulation runs. HDFS RPP generates uneven replica distributions in scenario one as confirmed in simulation results. The number of replicas assigned to one node Z spreads over a wide range from 8 to 70 in scenario one. HDFS RPP randomly selects nodes directly instead of selecting a rack of this node first, therefore, the replica distribution is not significantly affected by the node distribution across racks. However, if the rack is randomly selected, the replica distribution would be more uneven when the node distribution is heterogeneous, because the nodes of rack with less nodes would have more replica assignment load than the nodes of rack with more nodes. HDFS RPP produces more load on the rack that has fewer nodes because the selected rack for the first replica would be selected for the second replica too. On the other hand, our simulation confirms that the proposed IDPM can generate perfectly even replica distributions in scenario one because Each node has 30 replicas.

Algorithm 3: Replicas Distribution algorithm

Input: A collection of **L** racks each rack has $r_i(i = 0, 1, 2, , ...L-1)$ available nodes on it.

n data blocks to be distributed, which are numbered 1 through n.

Result: $T[0, 1, ..I-1][0, 1, .., N-1]$

```
1  begin
2      calculate N and I according to Eq. 3.2 and Eq. 3.3 respectively.
3      for i = 0 to L-1 do
4          Rack_compete [i] = r[i]; //initializing rack competition register.
5          for j = 0 to r[i] do
6              Node_free[i][j] = 0; // indicator to check if the node is assigned a block or
                 not.
7          end
8      end
9      rack_avoid = -1    // used to avoid the same rack for the third replica.
10     for Block = 0 to n-1 do
11         current_rack = Elected_rack(Rack_compete) ;
12         current_node = Elected_node(current_rack) ;
13         Count = 0;
14         for i=0 to current_rack do
15             Count=Count +r[i]; // to count the number of columns located before the
                 current rack.
16         end
17         Column =Count + current_node; //  calculated from Eq. 3.4
18         T [iteration[current_rack]][column]= Block;
19         current_node = Elected_node (current_rack);
20         Column = Count + current_node;
21         T [ iteration[current_rack]][column] = Block;
22         rack_avoid= current_rack;  //avoid current rack for 3rd replica.
23         current_rack = Elected_rack (Rack_compete)
24         current_node = Elected_node (current_rack)
25         for i=1 to current_rack do
26             Count=Count +r[i]; // to count the number of columns located before the
                 current rack.
27         end
28         T [ iteration[current_rack]][column]= Block;
29     end
30     return  T [0,1,..I-1][0,1,..,N-1]
31 end
```

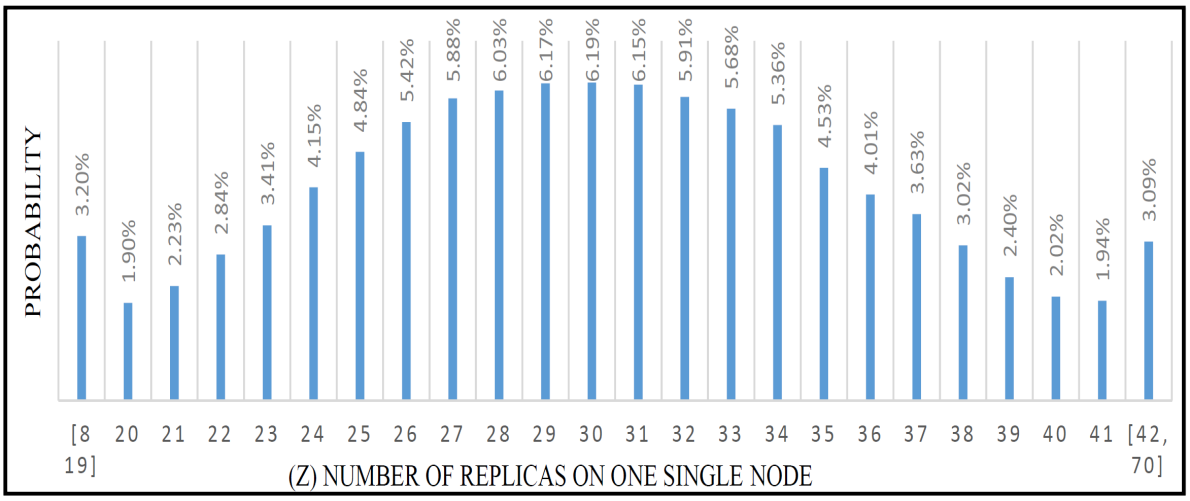

Figure 3.5: Probability distribution of Z in scenario one.

Because of the large scale of this scenario, a figure showing the replica distributions generated cannot be shown in this dissertation. Therefore, scenario two, reduced scale, is included to present the replica distributions generated by both HDFS RPP and IDPM as shown in Table 3.1. Fig. 3.6 shows one replica distribution generated by HDFS RPP in scenario two. The replica distribution is uneven with number of replicas on one single node ranging from 2 to 11. The replica distribution generated by IDPM in scenario two, as shown in Fig.3.7, is perfectly even, does not need to run balancing utility, and meets all HDFS replica placement requirements.

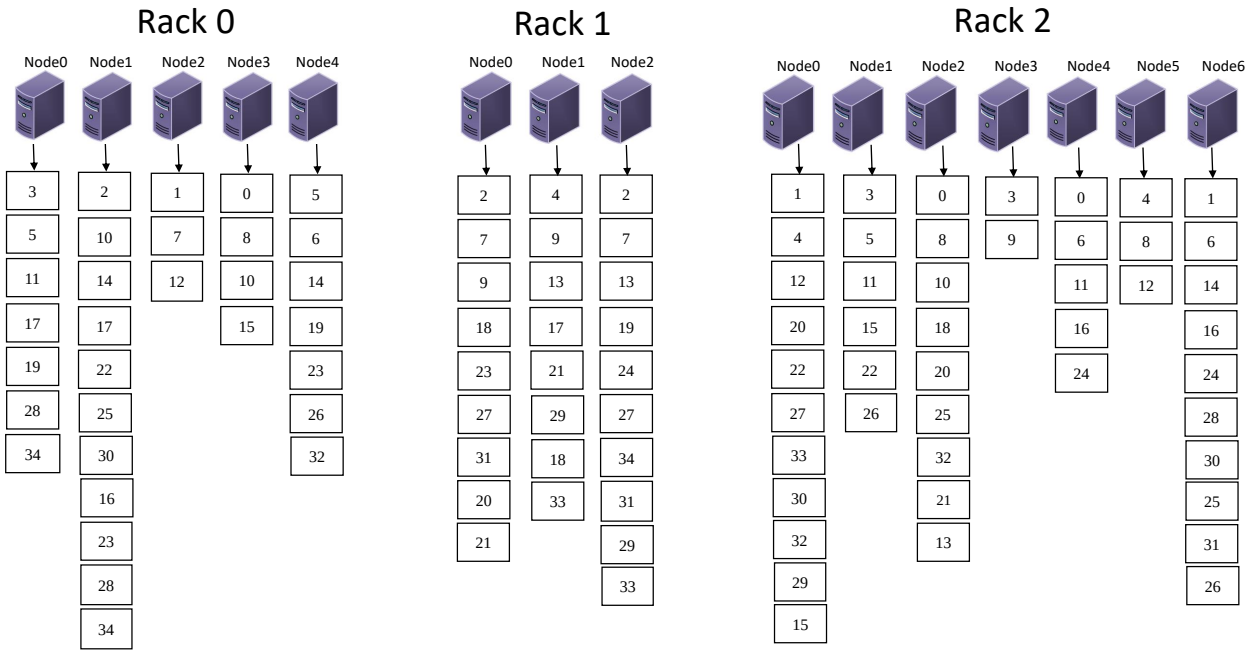

Figure 3.6: Replica distribution generated by HDFS RPP in scenario Two

Table 3.1: SIMULATION SETTINGS OF IDPM EVALUATION

Simulation Settings	Scenario One	Scenario Two
Total Number of Blocks	6,000	35
Duplication Factor	3	3
Total Number of Replicas	18,000	105
Total Number of Nodes	600	15
Average Number of Replicas on One Node	40	7
Minimum Number of Replicas on One Node	8	2
Maximum Number of Replicas on One Node	70	11
Total Number of racks	40	3
Number of available nodes on $rack_i$	15	5(i=0) 3(i=1) 7(i=2)

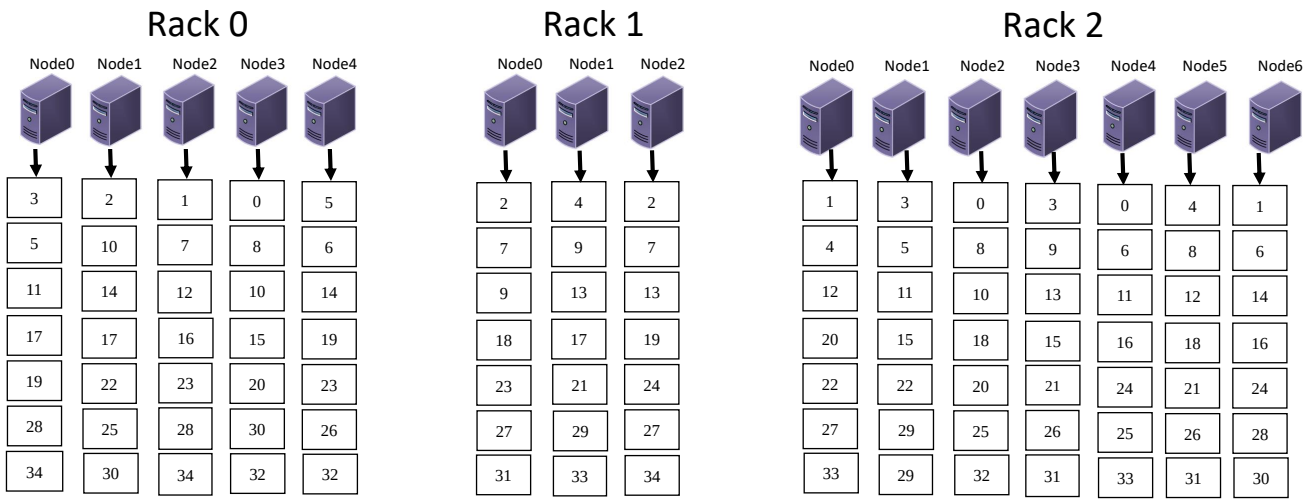

Figure 3.7: Replica distribution generated by IDPM in scenario two

CHAPTER 4: BALANCED REPLICA PLACEMENT POLICY FOR HETEROGENEOUS CLUSTERS[1]

Replica placement in HDFS plays a significant role in data availability and balancing the computational load on DataNodes. In the current Hadoop Distributed File System replica placement policy HDFS RPP the replicas of data blocks cannot be evenly distributed across the cluster of data nodes, specially when variety of hardware generations coexist in the cluster. As mentioned earlier in the previous chapter the current HDFS must rely on load balancing utility for balancing replica distribution, which results in extra overhead for time and resources. However, in heterogeneous clusters the even distribution of replicas does mean it is a fair distribution. It does not produce balanced computational loads on data nodes because of the variation in the processing capabilities and resources of data nodes. This chapter addresses the load balancing problem and presents an innovative replica placement policy for HDFS. It can perfectly balance the distribution of replicas of file among the data nodes in heterogeneous cluster environments. Experimental results of the proposed solution confirm that, the proposed replica placement scheme gives better DataNodes utilization than the default replica placement policy of Hadoop in heterogeneous cluster environment.

4.1 Introduction

In recent years, Hadoop framework is popularly known for providing cost-effective solutions to process large-scale data intensive applications in a distributed manner on cluster of processing nodes. Cluster nodes of Hadoop may vary in their processing capabilities and availability of resources. One of the key issues that can significantly affect the performance of cluster nodes in

[1]Related publication: I. A. Ibrahim and M. Bassiouni. Improvement of data throughput in data-intensive cloud com-puting applications. In2019 IEEE Fifth International Conference on Big Data ComputingService and Applications (BigDataService), pages 49–54. IEEE, 2019.

data intensive cloud computing is computational load balancing among cluster nodes. HDFS is the storage part of Hadoop framework. It is a file system designed for storing very large files reliably and stream data with high bandwidth, running on clusters on commodity hardware [68]. Cloud computing applications that perform massive computing tasks (big data processing) offload data and tasks to data centers or powerful servers in the cloud. In a typical Hadoop cluster, terabytes of data are processed using parallel computations. The processing of jobs in a Hadoop cluster is affected by the storage mechanism of data in HDFS, and the processing capabilities of cluster nodes [40].

4.2 Background

In our previous work [32], the need for running the balancing utility used by RPP for HDFS has been eliminated. An improved RPP for HDFS called Intelligent Data Placement Mechanism IDPM has been proposed for replica distribution in Cloud Storage Systems , IDPM distributes replicas evenly to cluster nodes and meet HDFS distribution requirements. However, IDPM works only for homogeneous cluster environments, while in heterogeneous cluster environments, where the racks are from different Hardware generations, the even distribution of replicas does not mean it is a fair distribution because it cannot guarantee the load balancing among cluster nodes. For example, in Fig 3.2. If the processing nodes of $Rack_1$ are slower than the processing nodes of $rack_2$, the even distribution of replicas produced by IDPM fails in balancing the processing load because it does not take into consideration the variations in processing capabilities of cluster nodes.

In HDFS, each data file is partitioned, replicated, and stored as a sequence of data blocks. In MapReduce jobs, DataNodes perform map tasks on the file's blocks in parallel. The current HDFS Replica Placement Policy (RPP) is not designed for heterogeneous clusters ,therefore, the replicas are not distributed over all nodes that selected to store the file based on their available resources. As a result, the highly utilized DataNodes experiences overload in client access, this

degrades the performance of MapReduce in heterogeneous environment. Similarly, the IDPM produces an even distribution of the file's blocks but in heterogeneous clusters environment this even distribution may lead to unbalanced processing load, thereby reducing the overall performance of Hadoop. Load balancing among heterogeneous cluster nodes would be a great enhancement for the performance of data-intensive cloud applications. It can improve job completion time, optimize the usage of resource, maximize the throughput, minimize the retrieval time, and minimize the overload on cluster nodes.

4.3 Speed-based Replica Assignment Policy SRAP

The current HDFS RPP generates unbalanced replicas assignment across all cluster nodes as shown earlier in Fig.3.2. Consequently, HDFS RPP runs a balancing utility for analyzing and re-balancing the replica placement among cluster nodes. In homogeneous cluster nodes, the data load balancing is achieved by equalizing the number of data blocks assigned to each node, while in heterogeneous cluster, the computational load balancing among cluster nodes cannot be achieved by merely balancing the number of data blocks assigned to each node. The number of blocks assigned to each node must be equivalent to the node processing capacity and available resources. The new policy proposed in this chapter generates balanced data distribution where the placement of data blocks is driven by the load on the nodes.

The proposed policy fairly distributes the data blocks to cluster nodes depending on their processing speed. The node that has higher processing speed is assigned more data blocks than another node with lower processing speed in order to get better node utilization. Consequently, increase the efficiency of data processing by cluster nodes to meet the requirements of data-intensive cloud computing. For example, during MapReduce jobs, if map tasks are applied on the blocks of many files that stored on the same cluster nodes simultaneously, the balancing of load among cluster nodes is achieved when the blocks have been distributed using the new policy because

nodes with high processing capacity have more data blocks than slower nodes. It minimizes the chance of overloaded nodes, eventually minimizes the job completion time. Since the key to the proposed replica placement policy is assigning replicas to nodes based on their processing capability, the new policy is Named Speed-based Replica Assignment Policy (SRAP). To deal with heterogeneous cluster environments, since the racks are usually belonging to different hardware generations, we assume the nodes on same rack are similar in their processing capability while nodes on different racks may differ in their processing capability. SRAP assigns a processing rank to the nodes. Rank of a node is the integer number of blocks it can process during a unit of time. The unit of time is the time needed by the slowest node in the cluster to process one data block. $Rank_1$ is the rank of the slowest node. However, SRAP works for homogeneous cluster environments too. It is a special case of heterogeneous cluster environments where the rank of all nodes is $Rank_1$.

4.3.1 Main assignment table MAT

Let's assume a cluster consists of L racks, each of which has r_i nodes, where $(i = 0, 1, \ldots, L-1)$. N is the total number of available nodes in the cluster. The nodes of each rack have a processing speed S_i where $(i = 0, 1, \ldots, L-1)$. Assume a file to be distributed, the file is partitioned into n blocks, the duplication factor, D, is assumed to be three replicas for each data block in this policy. The total number of replicas to be distributed to the cluster nodes is p, as calculated in Eq.4.1. The processing rank of nodes in $rack_i = K_i$, where $(i = 0, 1, \ldots, L-1)$. K_i is the greatest integer number less than or equal to the result of dividing the speed of nodes of $rack_i$ to the minimum speed in the cluster. It can be calculated by using Eq. 4.2.

$$p = n \times D \tag{4.1}$$

$$K_i = \frac{S_i}{\min\limits_{(0 \le k \le L-1)} S_k} \tag{4.2}$$

SRAP builds a rack assignment table for every rack, it is two dimensional array with m_i columns and I rows, every node has column(s) in the rack assignment table equals its processing rank K_i. In the proposed algorithm, a node in Rack i has K_i columns allocated for it in the rack assignment table. The number of columns in the assignment table of Rack i is m_i. It is calculated in Eq.4.3

$$m_i = r_i \times K_i \qquad for \;\; i = (0, 1, .., L-1) \tag{4.3}$$

The racks assignment tables are merged together to form the Main Assignment Table MAT. The number of columns in MAT is M. it is calculated in Eq.4.4

$$M = \sum_{i=0}^{L-1} m_i \tag{4.4}$$

SRAP build the MAT for every file in HDFS that need to be distributed across cluster nodes. So HDFS can use MAT of the file for the file's blocks distribution. The number of rows in MAT is I, it is calculated in the previous chapter using Eq. 4.5.

$$I = \left\lceil \frac{p}{N} \right\rceil \tag{4.5}$$

4.3.2 Block index allocation in MAT

SRAP builds the main assignment table MAT and distributes the blocks' numbers inside this table. Every field in MAT is corresponding an actual location in the cluster. So, HDFS can use MAT to distribute the actual blocks by placing each replica to the corresponding location referred by MAT.

SRAP has p replicas to be assigned to MAT. It starts the process of allocating the blocks' numbers, 0 through $n-1$, into the MAT. It only allocates M replicas at a time, row after row until it reaches to the last block's number. Every column in MAT is corresponding to a node in the cluster, one node may have more than one column. Columns that dedicated for node d in rack R start from column C through column $C+K_R-1$. C can be calculated according to Eq.4.6

$$C = d \times K_R + \sum_{i=0}^{R-1} (r_i \times K_i) \tag{4.6}$$

K_R is the rank of rack R. Starting from $Block_0$, for every block, the algorithm picks the row and columns to allocate it in the main assignment table. The row selection starts from row_0 to $row_{(I-1)}$. The selection of column must comply with the reliability constrains of RPP of HDFS. The column selection of SRAP consists of two phases: rack selection phase, and node selection phase. In rack selection a rack section is selected. While in node selection phase a free field in a column of a node inside this rack section is selected.

The rack selection, and node selection are the core of this policy. In rack selection phase, SRAP uses an index called Rack_Index to select two racks sections with more free fields on it. The index is continuously getting updated. Rack_Index of a specific rack is the number of free columns in this rack section for the current row. Node selection phase starts after rack selection. In this phase, a node with more free columns from the selected rack in the current row is selected. When every node is assigned replicas equal to the number of columns it has, a new row of distribution begins.

For example, let's assume a cluster of three racks, $Rack_0$ has two nodes, $Rack_1$ has three nodes, and $Rack_2$ has four nodes. The processing ranks are $(K_0) = 3$, $(K_1) = 1$, and $(K_2) = 2$. Then nodes of $Rack_1$ are the lowest processing capacity among the other nodes while nodes of $Rack_0$ are the highest processing capacity. As shown in Fig.4.1, SRAP generates three tables, the assignment table of $rack_0$ consists of six columns, the assignment table of $Rack_1$ consists of three columns,

and the assignment table of $Rack_2$ consists of eight columns. The main assignment table MAT is formed from merging the three tables, it has M = 17 columns and I rows, where M and I can be found according to Eq.4.4,and 4.5 respectively. MAT is shown in Fig.4.2. Let's assume a file consists of 22 blocks, every block is replicated to three replicas. Blocks' numbers are assigned to the cells of MAT 17 blocks per row, with keeping the same reliability constrained of replica placement policy for HDFS. The process starts with $Block0$, where two replicas of $Block0$ are assigned to $Rack_2$, each of these two replicas are placed on different node. $Rack_2$ is selected at the beginning because it is the rack that has more free nodes spots. The third replica of $Block0$ is assigned on another rack that has higher free node spots than the other racks, it is $Rack_0$. When all the columns of the first row are assigned blocks' numbers, the second row starts and so on until all blocks' numbers are distributed.

Finally, SRAP results in balanced and fair distribution of replicas to cluster nodes in term of processing capabilities for better utilization of all the cluster nodes. SRAP is perfect replica placement algorithm for heterogeneous cluster environment. the pseudo code of the replica distribution algorithm is shown in Alg. 4, and 5. During the rack selection for the third replica, the algorithm must avoid the rack that store the first and second replicas. To do so, SRAP tags the rack on which the first two replicas of current block reside as *avoid_Rack* in order to be discarded during the process of finding the elected rack for the third replica. The Algorithm used to return the elected rack is shown in Alg. 6. This Algorithm is used two times in the main algorithm. First to place the first two replicas, and second, to place the third replica. During the algorithm of electing a node for the selected rack, Alg. 7, SRAP tags the node on which the first replica of current block resides as *avoid_Node* to discard it during the process of finding the elected node for the second replica. After each rack election, SRAP assigns the replica to the elected node of this rack. The process continues until the completion of distribution of all replicas.

Algorithm 4: Replicas Distribution algorithm using SRAP

Input: A collection of **L** racks each rack has $r_i(i = 0, 1, 2, , ...L-1)$ available nodes on it.
n data blocks to be distributed, which are numbered 0 through $n-1$.
Rank of every node in $Rack_i$ is K_i, where (i= 0,1,2, ..., L-1)
Result: $T[0, 1, ..I-1][0, 1, .., N-1]$

```
1  Begin
2      calculate M and I according to Eq. 4.4 and Eq. 3.3 respectively.
3      for i = 0 to L-1 do
4          rack_compete [i] = r[i] ×K[i]; //initializingrackcompetition.
5          for j = 0 to r[i]-1 do
6              node_free[i][j] = K[i]; // indicator to check if the node is assigned a block
                 or not.
7          end
8      end
9      rack_avoid = −1
10     for Block = 0 to n-1 do
11         current_rack = elected_rack(Rack_compete) ;
12         current_node = elected_node(current_rack) ;
13         Count = 0;
14         for i=0 to (current_rack − 1) do
15             Count=Count + r[i] ×K[i];
16         end
17         column =Count + current_node ×K[current_rack];
18         for p=0 to (K[current_rack] − 1) do
19             if (T[iteration[current_rack]][column] = 0) then
20                 T[iteration[current_rack]][column+p] =Block;
21                 Exit the loop;
22             end
23         end
24         current_node = elected_node (current_rack);
25         column = Count + current_node × K[current_rack]; for p=0 to
             (K[current_rack] − 1) do
26             if (T[iteration[current_rack]][column] = 0) then
27                 T[iteration[current_rack]][column+p] =Block;
28                 Exit the loop;
29             end
30         end
31     end
```

Algorithm 5: Replicas Distribution algorithm using SRAP- Part 2

```
rack_avoid= current_rack;
current_rack = elected_rack (Rack_compete);
current_node = elected_node (current_rack);
for i=0 to current_rack-1 do
    Count=Count +r[i] ×K[i];
end
column = Count + current_node × K[current_rack];
for p=0 to K[current_rack] − 1 do
    if (T[iteration[current_rack]][column] = 0) then
        T[iteration[current_rack]][column+p] =Block;
        Exit the loop;
    end
end
return T [1,2,..I][1,2,..,N]
```

Algorithm 6: FUNCTION - Elected rack using SRAP

```
Input: Rack_compete[0,1,..R − 1]
Result: current_rack
begin
    if (rack_avoide = 0) then
        current_rack =1;
    end
    for i = 0 to L-1 do
        if (i = rack_avoid) then
            skip the current rack for the third replica;
        end
        if iteration[i] < iteration[current_rack] then
            current_rack = i; // use the rack with the lowest iteration.
        end
        if rack_compete[i]>rack_compete[current_rack] then
            current_rack= i;
        end
    end
    return current_rack
end
```

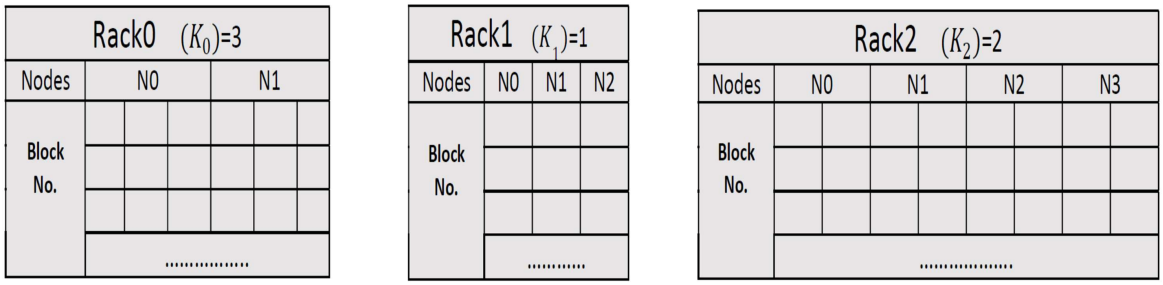

Figure 4.1: Example of racks assignment tables

Rack	0						1			2							
Node	0			1			0	1	2	0		1		2		3	
	Main Assignment Table																
Block No.	0	1	3	4	6	7	1	1	2	0	2	0	3	3	2	4	5
	4	6	8	9	10	11	5	6	12	5	7	7	8	8	9	9	10
	10	12	14	12	14	15	16	13	13	11	13	11	14	15	16	15	16
	17	18	19	17	19	20	21	21		17	18	18		20	20	21	19
	.....						.....			.....							

Figure 4.2: Main Assignment Table MAT using SRAP

4.4 Evaluation

The simulation results confirm that SRAP can generate perfectly even and fair replica distributions of replicas in term of data node processing capability. Two scenarios are considered; in each scenario an experiment was conducted as shown in Table 4.1. Scenario one represents heterogeneous environment like the previous example in Fig.3.2. Three different generations of hardware exist: nodes in $rack_0$ are of rank 3, nodes in $rack_1$ are of rank 1, and nodes in $rack_2$ are of rank 2.

Algorithm 7: FUNCTION - Elected node using SRAP

```
Input: current_rack
Result: current_node
begin
    for i = 0 to r[current_rack]-1 do
        if i = node_avoide then
            Exit loop for for this node;
        end
        if (node_free[current_rack][i] > 0) then
            current_node = i;
            node_free[current_rack][i] = (node_free[current_rack][i])-1;
            Exit and End the for loop;
        end
    end
    if (rack_compete[current_rack] = 0 ) then
        iteration[current_rack]= iteration[current_rack]+1;
        rack_compete[current_rack]= r[current_rack]*K[current_rack];
        for i = 0 to r[current_rack]-1 do
            node_free[current_rack][j] = K[current_rack];
        end
    end
    return current_node
end
```

In replica distribution generated by the HDFS RPP, the replicas are not evenly distributed across the cluster nodes, and there are nodes with low computing capabilities are assigned replicas more than other nodes with high computing capabilities. It results in node overload and significant delay in data processing as shown in Fig.3.2. For example, $node_2$ of $rack_1$, although it has low processing speed, it has been assigned 14 data blocks, while $node_0$ in $rack_0$ that has higher processing speed is assigned 9 data blocks only. Even though HDFS user the balancing utility, it will not take into consideration the heterogeneity of cluster.

On the other hand, replica distributions generated by the SRAP generates balanced replicas distribution based of the processing capabilities of the node, as shown in Fig 4.3. Nodes of rack 0 have been assigned more replicas than Nodes of rack1. Scenario two represents a highly heteroge-

neous environment where there exist four different generations of hardware: nodes on $Rack_0$ are rank 1, nodes on $Rack_1$ are rank 2, nodes on $rack_2$ are rank 3, and the node on $Rack_3$ are rank 4. In Scenario two, the number of replicas to be distributed to 11 nodes are 192 replicas, every column in MAT has 8 replicas. The results of scenario two are shown in Table 4.1.

Table 4.1: SIMULATION SETTINGS OF SRAP EVALUATION

Simulation Settings	Scenario One	Scenario Two
Total Number of Blocks.	34	64
Duplication Factor.	3	3
Total Number of Replicas.	102	192
Total Number of Racks.	3	4
Total Number of nodes in each rack.	$r_i = [2,3,4]$	$r_i = [4,3,2,2]$
Rank of node in each rack.	$K_i = [3,1,2]$	$K_i = [1,2,3,4]$
Number of replicas on one column.	6	8
Minimum number of replicas on One node	6	8
Maximum number of replicas on One node	18	32

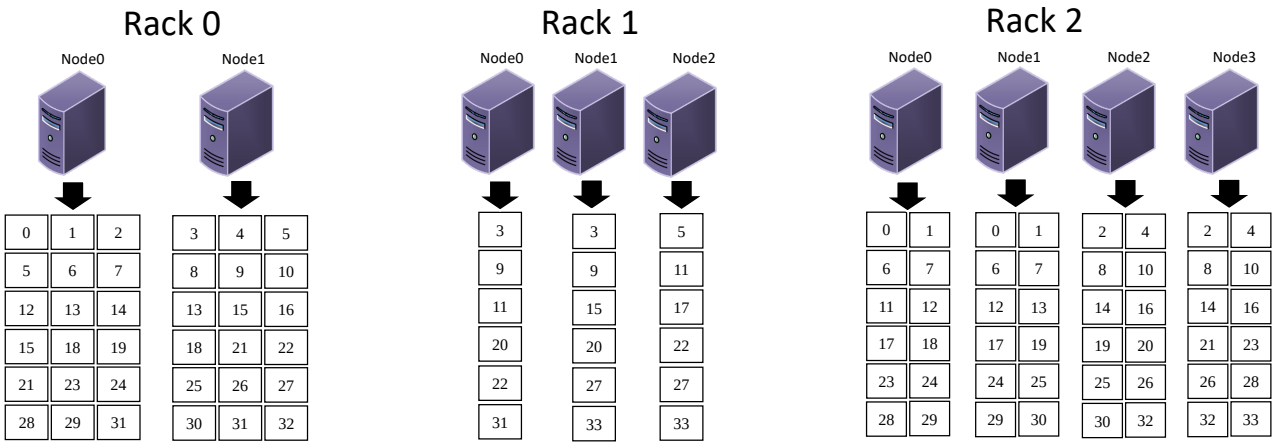

Figure 4.3: Replicas distribution generated by SRAP

CHAPTER 5: STRAGGLER IDENTIFICATION AND SPECULATIVE EXECUTION[1]

Straggler tasks dramatically impede parallel job execution of data-intensive computing in Cloud Data centers. However, data-intensive computing frameworks, such as MapReduce or Hadoop, employ a mechanism called speculative execution to deal with the straggler issue. Speculative execution method is a widely adopted as a straggler identification and mitigation scheme. However, speculative execution provides limited effectiveness because in many cases straggler identification occurs too late within a job life cycle, or unsuccessful backup tasks are initiated based on inaccurate straggler identification process. The successful identifying of the straggler, and the timing of identifying it are very important for straggler mitigation in data-intensive cloud computing. In this chapter, a new straggler identification scheme is proposed to make Hadoop more efficient in cloud environments. It is named Progress and Feedback based Speculative Execution (PFSE) algorithm. This algorithm identifies the straggler Map tasks based on the feedback information received about the completed tasks and the progress of the current processing task. The extensive experiments show that PFSE can outperform dynamic scheduling techniques like Self-Learning MapReduce scheduler (SLM) and Longest Approximate Time to End (LATE) algorithm. PFSE can assist in enhancing straggler identification and mitigation for tolerating late-timing failures within data intensive cloud computing.

5.1 Introduction

Hadoop is an open-source implementation of MapReduce, it has been applied in data parallel processing on a large cluster of commodity machines to handle large-scale data intensive applications.

[1]Related publication: I. A. Ibrahim and M. Bassiouni. Improving mapreduce performance with progress and feed-back based speculative execution. In 2017 IEEE International Conference on Smart Cloud(SmartCloud), pages 120–125. IEEE, 2017.

It divides a large computation job into small tasks and assigns them to multiple computational cluster nodes running in parallel. It can be scaled easily to large clusters of inexpensive commodity computers, automatically handles failures, and hides the complexity of fault tolerance from the programmer. The attractive feature of MapReduce is the ability to automatically divide a job into multiple tasks and transparently handle tasks execution by distributed processing nodes [30]. In a MapReduce cluster, after a job is submitted, the job is divided into multiple map and reduce tasks. MapReduce works by breaking the processing into two phases: the map phase and the reduce phase. Map tasks and the reduce tasks are distributed into processing nodes which continue processing the tasks and keep updating the tasks' progress by periodic heartbeat. Map tasks extract (Key, Value) pairs from the input data, transfer them to some user defined map function and combine function, then generate map outputs. The output from the map function is processed by the MapReduce framework before being sent to the reduce function. This process sorts and groups the (Key, Value) pairs by key, transfers the stream to some user defined reduce function, and finally generates the result of the job [16].

In general, a map task is divided into map and combine phases, while a reduce task is divided into copy, sort and reduce phases. Since Hadoop-0.20, reduce tasks can start whenever some map tasks are completed, to avoid network congestion because the reduce tasks copy map outputs whenever they become available without waiting for all the map tasks to be completed. However, reduce task can start the sort phase when all map tasks have been completed. This is because each reduce task must finish copying outputs from all the map tasks to make the input ready for the sort phase to produce the final result. This chapter focuses on straggler identification of tasks in map and sort phases.

5.2 Background

In MapReduce, after a job is submitted, the NameNode divides the input files into multiple map tasks, and then schedules both map and reduce tasks to the DataNodes. The DataNode runs tasks in its containers and keeps updating the tasks' progress to the NameNode by periodic heartbeat. Input data is partitioned and distributed to the computing nodes in the map phase. The intermediate data generated in this phase are sorted then transferred to the nodes that perform reduce process. Hash-Partitioner is the default partitioner of Hadoop that used to partition the intermediate data to be submitted to the reducers [21]. One of the significant issues that can delay the final result of MapReduce job is the Straggler task. Data-intensive computing frameworks, such as MapReduce or Hadoop, employ a mechanism called speculative execution to deal with the straggler issue. It runs a speculative copy of a straggler's task on another DataNode. The default speculative execution algorithm of Hadoop aggressively starts many backup tasks in order to avoid straggler tasks. However, the current speculative execution algorithm of Hadoop has limited effectiveness because of the unsuccessful backup tasks. The inaccuracy in identifying the straggler causes initializing a number of unnecessary backup tasks which degrade the performance of MapReduce and leads to the increasing of MapReduce job completion time.

5.3 Progress and Feedback based Speculative Execution (PFSE) algorithm

In this dissertation, an algorithm named Progress and Feedback based Speculative Execution (PFSE) is developed in order to improve MapReduce performance. This algorithm reduces the number of unnecessary backup tasks by improving the estimation of the task execution time. PFSE calculates phase completion time estimation from the fresh information collected from recently completed tasks of the same job, and the current progress of task execution during the phase. To evaluate the performance of PFSE, it has been implemented on Hadoop computing environment, and experiments with four MapReduce application jobs have been conducted. the performance of

PFSE has been compared with the performance of those of existing algorithms. PFSE focus only on straggler tasks in map phase, because running backup tasks for map tasks generally does not require exchanging intermediate data between nodes. However, a policy for straggler mitigation in reduce phase is proposed in the next chapter. PFSE algorithm estimates the remaining execution time of each task. It uses the progress rate from the current phase and the feedback from the completed phases to estimate the completion time of running tasks.

In PFSE, the accuracy of the estimation increases as the completed tasks increase. Two factors dramatically combined to give better estimation about the task completion time. First factor is the current phase Progress Score (P_s) of the task. P_s is the amount of data the current task processes during the current phase. the value of P_s is continuously updated. The larger the processed data, the more precisely the estimated remaining time by using this factor. The second factor is the feedback that has been received about the completion time of the already completed phases of tasks from the same job. The larger number of completed tasks of the same job, the more accuracy estimation of completion time using this factor. The accuracy of these factors depending on the time during the job processing cycle. Calculations from completed phases information and that from currently running tasks play significant role to accurately estimate the completion time. The role of the source of calculations is changing over time depending of the accuracy of each calculation. The second factor is used only in case of there is at least one completed task. In the early execution stages, where there is no completed task, the only factor used is the P_s.

At the beginning, the algorithm calculates the estimated completion time $ECT[i]$ of the currently running $task_i$, where there is no finished task yet of the job including $task_i$ and the task currently in the map phase. The algorithm estimates the ratio of execution time of the map phase to that of the sort phase for the current $task_i$. At the beginning, the default value of Ratio is $2/3$,it is similar to the default phase weight in LATE. Let's assume ps_i is the progress score of $task_i$, which can be obtained from Hadoop, and ranges from zero to one. For each $task_i$, ps_i is calculated as the ratio of the amount of data successfully processed in $task_i$, to the amount of total data to be

processed by $task_i$.

During the map phase of $task_i$ the remaining time of the map phase, $T_{rm}[i]$, at the current time $T_{cr}[i]$ can be estimated as in Eq.5.1. The $T_{cr}[i]$ is the duration of time from the first time that $task_i$ reports its map phase progress to the current time. Not all the duration from the beginning is taken, however the initial delay at the beginning of the phase is ignored in order not to negatively effect on the estimation. The next phase is the sort phase. The estimated execution time of the sort phase, $T_{sort}[i]$, for $task_i$ can be calculated in Eq.5.2. $T_{map}[i]$ is total map phase estimated time, it can be calculated using Eq. 5.3.

$$T_{rm}[i] = \left(\frac{1 - ps[i]}{ps[i]}\right) \times T_{cr}[i] \tag{5.1}$$

$$T_{sort}[i] = \left(\frac{T_{cr}[i]}{ps[i]}\right) \times \left(\frac{1 - Ratio}{Ratio}\right) \tag{5.2}$$

$$T_{map}[i] = \left(\frac{T_{cr}[i]}{ps[i]}\right) \tag{5.3}$$

Ratio is the ratio of map phase to sort phase. When the task is in map phase, the estimated completion time of the currently running $task_i$ is calculated in Eq. 5.4.

$$TotalT_{rm}[i] = TM_{rm}[i] + T_{sort}[i] \tag{5.4}$$

The Second factor that can enhance these calculations and increase the efficiency of backup task mechanism is the feedback from the completed phases. Every completed task is used to be used in the estimated completion time of the map and sort phases. Let's assume S is a set of completed tasks. The average execution time of map phase for the completed task, M_{avg}, and the

average execution time of sort phase for the completed task, S_{avg}, can be calculated as in Eq. 5.5, and 5.6 respectively.

$$M_{avg} = \frac{1}{N} \times \sum_{i \in S} M_{comp}[i] \tag{5.5}$$

$$S_{avg} = \frac{1}{N} \times \sum_{i \in S} S_{comp}[i] \tag{5.6}$$

N is the number of tasks in set S. $M_{comp}[i]$ is the execution time of the map phase of the i_{th} completed task in S. $S_{comp}[i]$ is the execution time of the Sort phase of the i_{th} completed task in S. The default value of ratio of map to sort is 2/3. However Ratio gets updated based on the received feedback from the competed tasks too. The average of map ratio, M_{Ratio}, and the average of sort Ratio, S_{Ratio}, are calculated in Eq. 5.7, and 5.8 respectively.

$$M_{Ratio} = \frac{M_{avg}}{M_{avg} + S_{avg}} \tag{5.7}$$

$$S_{Ratio} = \frac{S_{avg}}{S_{avg} + M_{avg}} \tag{5.8}$$

The estimated time of map phase can be adjusted based on the feedback received from the completed task. At the beginning, when there is no task completed, the default value of the estimated map task time, M_{def}, can be initially calculated from the speed of node that process the map $task_i$, $S_n[i]$, and the amount of data, M, that need to be processed in the map task. M_{def} is calculated in 5.9.

$$M_{def}[i] = \frac{M}{S_n[i]} \tag{5.9}$$

M_{def} is a default value. it assumes that the node is not busy and does not have any resource contention or bandwidth restriction for the processed data. This value gets adjusted if there is at least one completed task but its accuracy increases based on how many tasks have completed the map phase. As shown in Eq.5.10, the value of feedback of the map phase processing time, $M_{fb}[i]$, is either the default value, $M_{def}[i]$, or the average execution time of map phase for the completed tasks belonging to the same job,M_{avg}.

$$M_{fb}[i] = \begin{cases} M_{def}[i], & \text{if no map task has completed yet .} \\ M_{avg}, & \text{otherwise.} \end{cases} \tag{5.10}$$

Similarly, the feedback of sort phase processing time for $task_i$, $S_{fb}[i]$, can be either the default value, $S_{def}[i]$, if there is none of the task has finished the sort phase yet, or the average execution time of the sort phase for the completed tasks belonging to the same job. $S_{fb}[i]$, and $S_{def}[i]$ are calculated in Eq.5.11, and 5.12 respectively.

$$S_{def}[i] = M_{def} \times \frac{1 - Ratio}{Ratio} \tag{5.11}$$

$$S_{fb}[i] = \begin{cases} S_{def}[i], & \text{if non of map task or sort task has completed yet .} \\ M_{Avg} \times \frac{1-Ratio}{Ratio}, & \text{at least one map task has completed but not sort phase yet.} \\ S_{avg}, & \text{otherwise.} \end{cases} \tag{5.12}$$

The value of $Ratio$ is 2/3, it is the default ratio as mentioned earlier, but it gets updated when the values of M_{Ratio} and S_{Ratio} are calculated. Therefore, $Ratio$ in calculated as shown in

Eq.5.13

$$Ratio = \begin{cases} \frac{S_{Ratio}}{M_{Ratio}}, & \text{If at least one Map task and one Sort task have been completed.} \\ \frac{2}{3}, & \text{Otherwise.} \end{cases} \quad (5.13)$$

During the map phase, the estimated remaining processing time to complete both map and sort phases, $TotalT_{rm}$, can be calculated from the summation of the estimated remaining time for map phase, and the total estimated time of the sort phase based on the two factors Progress Score and the Feedback as shown in Eq. 5.14

$$TotalT_{rm}[i] = ps[i]T_{rm}[i] + fb[i](|M_{fb}[i] - T_{cr}[i]|) + ps[i]\frac{T_{map}[i](1-Ratio)}{Ratio} + fb[i]S_{fb}[i]. \quad (5.14)$$

Ps_i ranges from zero to one. The feedback weight has a reverse relation with the progress score. The initial estimations depends on the feedback because of the low progress score of the task. when the progress score is increasing, the reliability on the feedback getting lower, and the dependence on the data received about the processing progress of the task gets increased until the progress score approaches one.

$$fb[i] = 1 - ps[i] \quad (5.15)$$

The difference between the total map phase time estimated from feedback and the current time is the estimated remaining time for map phase based feedback. The absolute value is taken just to avoid a rare case when the current time is greater than the feedback expected map phase time. When $task_i$ starts the sort phase, the value of ps[i] is zero. The value of ps[i] starts increasing gradually based on the progress score in the sort phase. The same two factors are used to find the

estimated remaining time to complete the sort phase $T_{Srm}[i]$ using Eq.5.16.

$$T_{Srm}[i] = (1 - ps[i]) \times T_{cr}[i] + fb[i] \times (|S_{fb}[i] - T_{cr}[i]|) \tag{5.16}$$

$T_{Srm}[i]$ is the estimated remaining time to complete sort phase when the current task is in sort phase. The feedback value of the estimated remaining sort time comes from the total sort phase estimated time calculated from the feedback minus the current time. As calculated previously, the absolute value is taken to the same reason mentioned earlier. At the beginning of the sort phase, few of the map phases have been completed already, which can help in increasing the accuracy of estimation of the sort phase remaining execution time. Therefore, the feedback during the beginning of sort phase is more reliable.

There are four cases in which this algorithm find estimated completion time of $task_i$ in Map phase or sort phase. First, when $task_i$ is in the map phase and there is no feedback information about any completed task belonging to the same job. Second, $task_i$ is in the map phase and there is a feedback information. Third, $task_i$ is in the sort phase and there is no feedback information. Fourth, $task_i$ is in the sort phase and there is information collected about completed phases. Eq.5.14 calculates the estimated completion time of $task_i$ for the first and second cases, while Eq.5.16 calculates the estimated completion time of $task_i$ for the third and fourth cases.

5.4 Evaluation

To test the performance of the proposed strategy, a practical environment has been prepared. Four practical MapReduce application jobs based on HiBench benchmark suite: Wordcount job, KMean clustering job, PageRank job, and Inverted Index job were chosen to evaluate the performance of straggler identification strategy. The accuracy of the suggested algorithm is examined by comparing the estimated execution time of map task from each job with the estimations resulted from LATE and SLM algorithms.

The crucial factor is to minimize the number of the unsuccessful backup tasks resulted from implementing speculative execution of the tasks that tagged incorrectly as straggler. Based on the mechanism of the examined algorithms, the accuracy of the estimated completion time using the three algorithms gradually increases during the process time, the estimated execution time changes with the progress of task processing. However, the estimations of the PFSE become closer to the real value with the increasing of the completed map tasks and that what makes it different from the other algorithms.

The accuracy of the algorithm is measured by the difference between the estimated task completion time for each algorithm, and the real execution value, which is obtained after completing the task execution. The result from these estimations controls the decision of how many backup tasks must start speculative execution because the estimated completion time of it is higher than predefined value. The results of the experiments showed that, the three algorithms identified many tasks that should not have been identified as a straggler, but FBSE has the lowest number of unsuccessful backup Map tasks. So, the main factor of examining the algorithm is the successful backup rate. Table 5.1 shows the number of backup map tasks decided based on the estimated task completion time generated from each algorithm for each job type. The successful backup rate column shows the rate of successful backup map tasks initiated for speculative execution. As shown in the table, for the wordcount job, the number of initiated backup map tasks of LATE,SLM, and PFSE algorithms are 75, 63, and 51 respectively. PFSE outperforms LATE and SLM algorithms by minimizing the number of backup tasks, as well as, increasing the successful backup rate, which leads to improving MapReduce performance.

Table 5.1: COMPARISON OF LATE, SLM, AND PFSE ALGORITHMS

Job	Strategy	Number of backup tasks	Number of successful backup tasks	Number of unsuccessful backup tasks	Successful backup rate
Wordcount	LATE	75	42	33	56.00%
Wordcount	SLM	63	43	20	68.25%
Wordcount	PFSE	51	41	10	80.39%
KMean Clustering	LATE	86	55	31	63.95%
KMean Clustering	SLM	73	54	19	73.97%
KMean Clustering	PFSE	69	58	11	84.06%
PageRank	LATE	59	42	17	71.19%
PageRank	SLM	57	43	14	75.44%
PageRank	PFSE	52	44	8	84.62%
Inverted Index	LATE	86	51	35	59.30%
Inverted Index	SLM	77	49	28	63.64%
Inverted Index	PFSE	60	51	9	85%

CHAPTER 6: STRAGGLER REDUCE TASKS OF MAPREDUCE JOBS[1]

Data skew of intermediate data in MapReduce job causes delay failures due to the violation of job completion time. Data-intensive computing frameworks, such as MapReduce or Hadoop Yarn, employ HashPartitioner. This partitioner may cause intermediate data skew, which results in straggler reduce task. Straggler reduce task delays the final result of MapReduce job because the final result of reduce phase is computed after receiving the results of all reduce tasks including straggler reduce task. Therefore, the overall running time of a MapReduce job is determined by the longest running reducer. In this chapter, we strive to make Hadoop more efficient in cloud environments. A new partitioning scheme, called Balanced Data Clusters Partitioner (BDCP), is proposed in this chapter to mitigate straggler reduce tasks based on sampling of input data and feedback information about the current processing tasks. We examined the proposed partitioner BDCP in a real MapReduce job environment that generates big intermediate data with high data skew. BDCP has been compared with default Hadoop partitioner and another previously suggested partitioner . Our extensive experimental results show that BDCP can outperform the default Hadoop HashPartitioner and Range partitioner.

6.1 Introduction

Big data tools like Hadoop and Apache Spark provide productive high-level programming interface for large scale data processing. Hadoop uses MapReduce as programming paradigm. It has been used for parallel processing of large-scale data on a large cluster of commodity machines to handle data-intensive applications. The next generation of Hadoop, namely Hadoop YARN, is accommodated to various programming frameworks and capable of handling many kinds of workload such as interactive analysis, and stream processing.

[1]Related publication : I. Ibrahim and M. Bassiouni, “Improvement of job completion time in data-intensive cloud computing applications ”, in Journal of Cloud Computing, Springer Publishing 2019.

In a MapReduce cluster, the job is submitted, then it is divided into multiple map tasks. Map tasks extract (Key, Value) pairs from the input data chunks. All (Key, Value) pairs sharing the same key form a data cluster. The total number of (Key,Value) pairs in the data cluster is the data cluster size. Map outputs generate the intermediate data. The intermediate data are divided according to a user defined partitioner before being sent to the reducers [68]. Thus, the mapper groups the data clusters into partitions. The partition is a set of data clusters assigned to the same reducer. Therefore, the number of partitions in each mapper equals the number of reducers. Every partition from each mapper is sent to the corresponding reducer, as shown in Fig. 6.1. The default partitioner of Hadoop is HashPartitioner.

Since all map tasks use the same partitioner, all similar keys are dispatched to the same partition. Every partition consists of many data clusters. The number of data clusters is equal to the number of distinct keys in the input data. One reducer processes one data partition. Ideally, the resulted intermediate data consists of keys that are approximately similar in their values, and the reducers are not busy with other tasks. In this ideal case, all data cluster are similar in their sizes. Therefore, the data processing load on the reducers is balanced because all reducers process same number of data clusters. However, in real applications, the reducers vary in their assigned intermediate data because the data clusters vary in their sizes, and the reducers vary in their processing capabilities.

Moreover, one reducer may have been assigned too much data to process as compared to the other reducers for the same job, which results because of the data skew. Data skew refers to the unfairness in the amount of data assigned to each task. Consequently, the reducers complete their reduce tasks while heavy reducer becomes straggler reducer. The straggler reducers degrade the performance of MapReduce applications because the result of the reduce phase is computed after receiving the results of all reduce tasks including straggler reduce task. In cloud computing platform, the reduce task that receives extremely large data becomes straggler, eventually delays the overall job completion time.

Figure 6.1: Map Reduce process

Straggler problem is very common in reduce tasks in data-intensive MapReduce jobs because of three major reasons:

- The data skew resulted from the partitioner: In case of data skew, the resulted data load on a reducer is much higher than the data load on the other reducers for the same job. In Hadoop,

data skew happens because of the keys dispatching is based on the hashing algorithm. The size of partition depends on the number of relevant (Key, Value) pairs. Data skew is one of the main performance bottlenecks in MapReduce environment.

- The variations in computing capabilities of reducers: In heterogeneous Hadoop cluster the DataNodes may vary in data processing speed due to the diversity of their computing capabilities [58]. Even if there is no data skew, the variations in computing capabilities of DataNodes that perform the reduce tasks lead to the case in which the slow node becomes straggler.

- The network congestion: It is resulted from the huge amount of data transferred from mappers to the reducers during the shuffle phase. Since the reducer waits for all the data clusters to arrive in order to start reduce task, the delay in transferring the data needed by a reducer leads to straggler reduce task.

Partitioner controls the partitioning process of the keys generated in map phase. The key (or a subset of the key) is used to derive the partition, typically by a hash function. The total number of partitions is the number of reduce tasks of the job. During the reduce phase, a data partition of large size may be assigned to one reducer while the other reducers receive small size partitions, as shown in Fig. 6.1. Consequently, all other reducers complete their reduce tasks, while the reduce task on the large partition becomes straggler because the result of this task takes long time, which leads to delay of final result.

6.2 Background

Hadoop 2.9.2 employs the following static hash function to partition the intermediat (Key, Value) pairs [13].

$$Hash\Big[HashCode\big(Key\big)mod\big(numReducer\big)\Big].$$

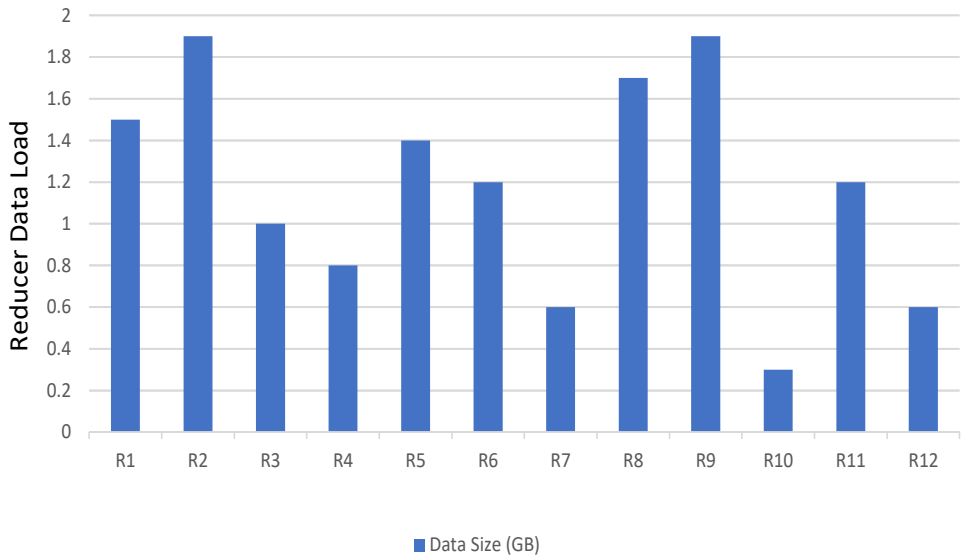

Figure 6.2: Variation in reducers data load

Unfortunately, this hash function used by Hadoop cannot solve the issue of skewed data. For reduce tasks, partitioning skew leads to shuffle skew, eventually some reducers will receive more data than others [76]. For example, Fig. 6.2 shows the different amounts of input data that have been assigned for 12 reducers when running the benchmark Word Count using 15 GB of text data [1].

Many straggler mitigation techniques have been developed in order to solve the issue of straggler. One of straggler mitigation techniques is the reallocation of straggler reduce task. In this technique, an alternative reducer is selected to run the reduce task. This process requires transferring all data of the reduce task to the new alternative reducer. In some cases, task reallocation leads to a higher overhead compared to the overhead produced by processing the task using the original slow node. This overhead resulted from the delay of transferring the data over the network to the alternative node.

However, the decision of transferring a reduce task to another reducer should be based on accurate calculations about the data transfer cost in order to minimize the completion time of the reduce phase. The ideal way to avoid straggler reduce tasks issue is by distributing the data clusters evenly as much as possible to the reducers. In order to distribute the intermediate data to reducers in an efficient way, the partitioning policy must be based on information about the (Key, Value) pairs resulted from each map task before the beginning of the shuffle process, and the data processing capabilities of reducers. The information must include the frequency of each key produced from every mapper in order to apply the partitioning policy that produce partitions that are similar in their sizes. Obtaining an ideal solution for this problem is unrealistic because of two reasons:

- In current Hadoop, the execution time of shuffling the mappers outputs to the reducers is overlapped with the map tasks execution time. Reduce phase is activated when specific percentage of map tasks have been completed, (5% by default in Hadoop 2.9.2). Overlapping the execution of map tasks and reduce tasks is handled in order to avoid network congestion, and fully utilize the resource. Consequently, minimizing the job completion time.

- An accurate information about the intermediate data can be obtained only when all map tasks have been finished. However, it is meaningless to obtain the (Key, Value) distribution after processing all input data in map phase, because the cost of pre-scanning the whole data is hard to be accepted when the amount of data is very large.

Furthermore, when the amount of input data is very large, the job completion time in case of waiting for all map task to finish then starting the shuffling is higher than the job completion time when the current default hash policy has been used [43].

6.3 Balanced Data Clusters Partitioner BDCP

An algorithm named Balanced Data Clusters Partitioner BDCP has been developed to improve MapReduce performance. This algorithm reduces the MapReduce job execution time by addressing the problem of straggler reducers caused from skewed data, network overhead, and slow reducers, by the following contributions:

1. Minimizing the effect of intermediate data skew.
2. Preventing the reducers skew by balancing the data load on the reducers.
3. Minimizing the amount of data transfer during shuffle phase over the network from mappers to the reducers.

The main steps of this algorithm are summarized as following:

1. Implementing the MapReduce job on a small sample from each split of the input data.
2. Calculating the estimated frequency of every key .
3. Collecting feedback information about the computing capabilities of reducers.
4. Building a new partitioning policy of the intermediate data for heterogeneous and homogeneous reducers nodes.

BDCP policy starts the sampling phase before the actual execution of MapReduce job. In sampling phase, a sample from the data input of each map task is taken. The sampling process used must ensure an accurate data representation of the original data in the sample data. The original MapReduce job is applied on the sampled data by using the same mappers and reducers reserved for this job. Once the information about the intermediate data and the reducers processing capabilities are received by the partitioning algorithm, the partitioning policy is created, and the MapReduce job starts to be implemented on the original data.

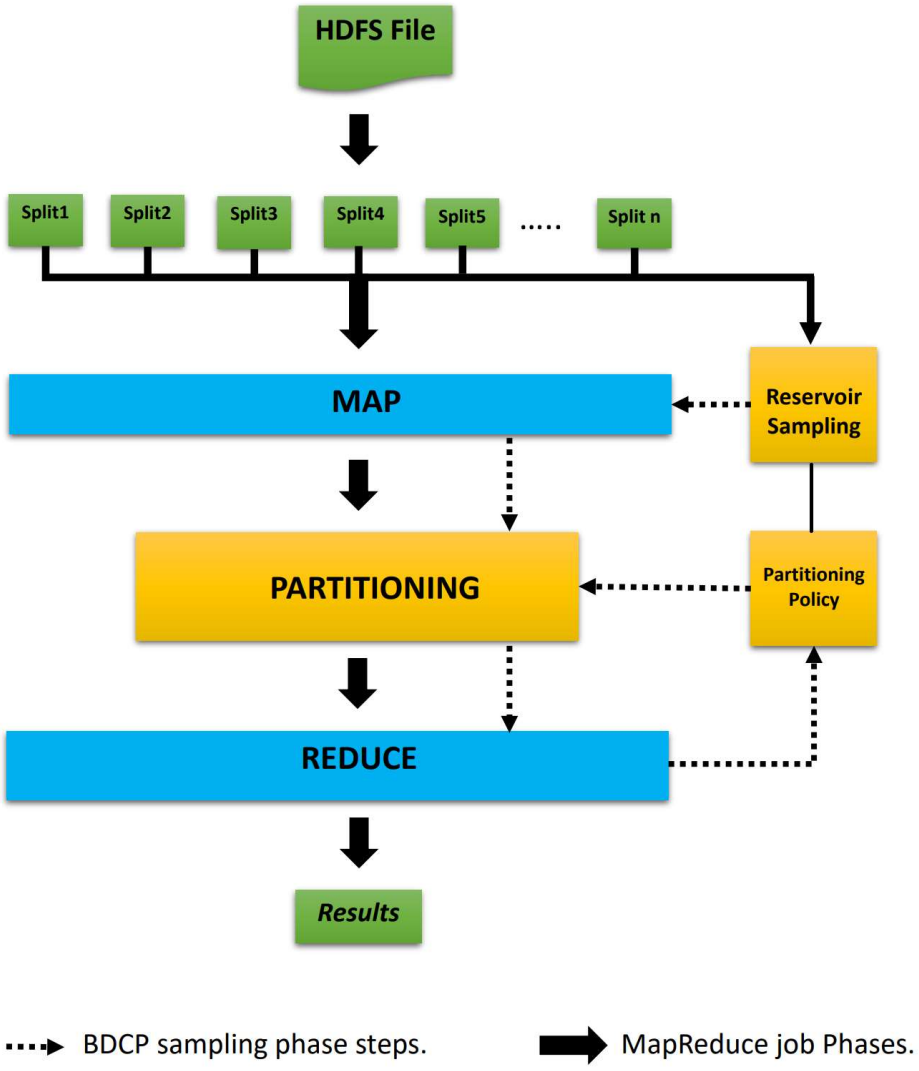

Figure 6.3: MapReduce job with BDCP partitioning policy

The efficiency of the algorithm depends on the size of data sample, the accuracy of sampling method used, and the accuracy of feedback information about the reducer nodes. The bigger the size of data sample used, the more accuracy of the estimation of the key frequency for the original data.

However, the bigger size of data sample used, the longer time it takes to complete the sampling phase, consequently longer job completion time. Sample size %10 of the original input data is used in the experiments in policy evaluation. The main processing steps of MapReduce job with BDCP partitioning policy is shown in Fig. 6.3. At the beginning, the reservoir sampling takes a sample of input data from each split using efficient sampling method as discussed in next section, then the regular MapReduce job is implemented on the sampled data. The intermediate data are partitioned using the default HashPartitioner. The reducers that are assigned to the original job is used to reduce the intermediate data in sampling phase. The last part of sampling phase is that the partitioning policy part of BDCP receives the results. However, the NameNode receives the information about speed of data processing of the reducers through the heartbeats. Thus, during the sampling phase and before the actual execution of MapReduce job, an accurate information about the data processing tare of the reducers are available in the NameNode to be used in the distribution policy. The distribution policy uses a modified knapsack problem algorithm. It assumes the reducers are the buckets with one size or different sizes, and data clusters are the items that need to be placed inside the buckets. The size of the reducer is based on the data consumption rate received by NameNode.

6.3.1 Sampling

Sampling is the selection of a subset (representative sample) from a target population then collecting data from that sample in order to estimate characteristics of the whole population. It is an efficient tool to reduce the amount of input data and dealing with a sample as a representative for the original data. Even though there are many sampling techniques, the type and features of data determine the sampling method that makes best representation of the original data. For example, if the data set is already sorted and a sampling method needed to find the distribution of this data set, then the best sampling technique is the interval sampling method. When a general information about the density distribution of numbers, and the probability about data distribution are known,

then a probability sample may be used.

In probability sample every unit in the population has a chance of being selected in the sample, and this probability can be accurately determined. When the data set is entirely unknown, it is best to apply simple random sampling. In a simple random sample (SRS) of a given size, all such subsets of the frame are given an equal probability. Each element of the frame thus has an equal probability of selection. Since the Hadoop MapReduce processes different kind of data with different features, it is best to use simple random sample. In practical applications of Hadoop MapReduce, the amount of data is very large, therefore BDCP use the sampling phase. The number of (Key, Value) pairs for each key in sample is approximately the proportion of (Key, Value) pairs in the original input data. The number of (Key, Value) pairs sharing the same key appear in the sample can be scaled up by dividing it by the sampling ratio to produce the estimated frequency of this Key in the original data, as in Eq. (6.1).

$$size(k) = \frac{size(k^{'})}{S\_Ratio} \tag{6.1}$$

S_Ratio is the sampling ratio, k is the original key, and $k^{'}$ is the key that appears in the sample.

6.3.2 Reservoir sampling

The reservoir sampling takes k elements from the population. It saves k preceding elements first, then randomly replaces original selected element in the reservoir with a new element that is selected from outside the reservoir. The final sample data of size k is generated after finishing the scanning of all the original input data, as shown in Alg. 8. Assume S is the data population and the required sample size is k. The algorithm creates a "reservoir" array of size k, and directly place first k items of S in it. It then iterates through the remaining elements of S, beginning from the $(k+1)^{th}$ element

until the last element of S. At the i^{th} element of the iterations, the algorithm generates a random number j between 1 and i. If j is less than or equal to k, j^{th}element of the reservoir array is replaced with the i^{th} element of S. In effect, for all i, the i^{th} element of **S** is chosen to be included in the reservoir with probability k/i. Similarly, at each iteration the j^{th} element of the reservoir array is chosen to be replaced with probability $(\frac{1}{k}) \times (\frac{k}{i}) = (\frac{1}{i})$. The time complexity of Alg.8 is analyzed as follows: Line 2 takes O(1) time. The time of the loop in lines 3–12 depends on N, where the data blocks consist of N records to be assigned. Therefore, the time complexity of this loop is O(N). If statement in lines 4–12 take O(1) time. So the total time complexity of sampling algorithm is O (N) .

Theorem: When the Reservoir Sampling algorithm has finished sampling process on a data set, each item in the data set has gotten equal probability of being chosen for the reservoir.

Proof: Let's assume that a sample of size k representing data set of size S. We are required to prove that each item in S has gotten equal probability of being chosen for reservoir. As shown in Fig.6.4, assume the algorithm is in the $(i-1)^{th}$ round, x is the element of the $(i-1)^{th}$ round, it either selected as a sample in the reservoir or skipped. The probability of x is selected and being in the reservoir array after completing round $(i-1)$ is $(\frac{k}{i-1})$. Since the probability of the j^{th} element of the reservoir array is chosen to be replaced in the i^{th} round is $(\frac{1}{i})$, the probability that x survives inside the reservoir in the i^{th} round is $(\frac{i-1}{i})$. Thus, the probability that x is in the reservoir after the i^{th} round is the product of these two probabilities, i.e. the probability of being in the reservoir after the $(i-1)^{th}$ round, and probability of x staying inside reservoir in the i^{th} round: $(\frac{k}{i-1}) \times (\frac{i-1}{i}) = k/i$. The probability for the i^{th} element to be swapped in is also $\frac{k}{i}$. Hence, the result holds for i. Since the base case of $i-1=k$ is true, the result is true for all $i \geq k$ by induction. In general, for $(k+1) < i \leq n$, probability of $S[i]$ being in the reservoir is: (Probability of selecting $S[i]$ to be in the reservoir in i^{th} round)$\times$(Probability of not removing $S[i]$ from the reservoir during the $(i+1)^{th}$ round)$\times$ (Probability of not removing $S[i]$ from the reservoir during the $(i+2)^{th}$ round)$\times..\times$ (Probability of not removing $S[i]$ from the reservoir during the n^{th} round).

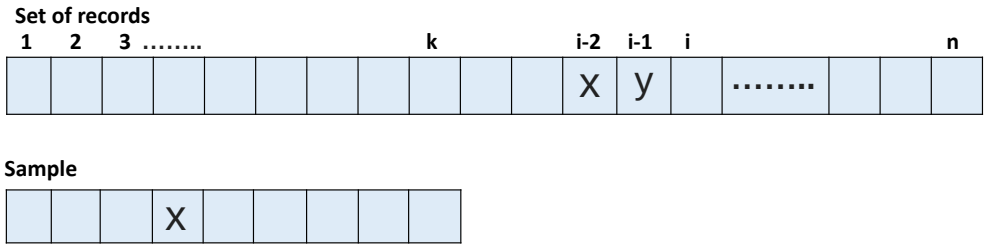

Figure 6.4: Reservoir sampling

The probability of $S[i]$ being in the reservoir can be simplified as follows:

$$p(i) = \frac{k}{i} \times \frac{i}{i+1} \times \frac{i+1}{i+2} \times \frac{i+2}{i+3} \ldots \frac{n-2}{n-1} \times \frac{n-1}{n} = \frac{k}{n}$$

the result is true for all $(k+1) < i \leq n$.

Algorithm 8: Reservoir sampling.

Input: Input: $\mathbf{S_i}[1,2,..,\mathbf{N}]$: Data block, it has **N** records.
Data blocks consist of **N** records to be assigned, which are numbered from 1 to (**N**)
Result: $R_i[1,2,..,K]$: sample which has k records.

```
1  begin
2      CB = 0; (Current block number)
3      for i = 1 to n do
4          if i ≤ k then
5              R[i] = S[i];
6          else
7              j = random(1, i);
8              if j ≤ k then
9                  R[j] = S[i];
10             end
11         end
12     end
13     Return R
14 end
```

6.3.3 Partitioning policy

Partitioning of the data clusters is the process of grouping data clusters into partitions. Every partition is assigned to a reducer to implement the reduce task. To increase the utilization of the resources in reduce phase, the reducers must be similar in their data load received during the shuffle phase. After finding an accurate sample that represents the original input data, partitioning policy of BDCP is decided. There are three important factors controlling the partitioning policy:

- Estimated $(key, value)$ distribution.
- Data processing rate of the reducers.
- Network bandwidth.

Let's assume that the number of mappers and reducers are M, and R, respectively. The key and value sets are K, and V respectively. The intermediate data set, $(key, value)$ pairs, generated during map phase is I. Data cluster is the set of all $(key, value)$ pairs that share the same key. The number of data clusters in I is C. The data clusters, $Cluster_1$ through $Cluster_c$, need to be distributed into R reducers.

$k_i \in K$ is the key of $Cluster_i$ for $i = 1, 2, ..C$.

$Cluster_i = <k_i, v> \in I$ for $i = 1, 2, ..C$.

One partition, P_i, is a set of data clusters that assigned to $Reducer_i$. Therefore, the intermediate data I consists of partitions from P_1 through P_R: $I = (P_1, P_2, .., P_R)$. Let's assume a Hadoop cluster consists of R reducers. C is the total number of data clusters appearing in the sampling phase. The frequency of keys in data $cluster_i$ is key_i , where $(i = 1, 2, ..., C)$. The total number of

$(Key, Value)$ pairs that need to be distributed to the reduce tasks is D. As in Eq. (6.2).

$$D = \sum_{i=1}^{C} key_i \tag{6.2}$$

Since data clusters vary in their sizes, and one data cluster must be assigned to one reducer, the sizes of partitions are not similar. Consequently, the distribution of data loads to the reduce tasks cannot be even. The mean size of the partitions, $Mean$, can be calculated as shown in Eq. (6.3).

$$Mean = \frac{D}{R} \tag{6.3}$$

The ideal case is when the data partitions are similar in their size, i.e. the partition size equals $Mean$. Practically this case is unrealistic, but as much as the partition size is close to $Mean$ as much as it is closer to the ideal case. The suggested partitioning algorithm minimizes this variation. the partitioning algorithm tries as much as possible to minimize the standard deviation of the number of $(Key, Value)$ pairs on every reducer from $Mean$, as shown in Eq. (6.4).

$$\begin{aligned} \min_{T_{ij}} \quad & \mathbf{s} = \sqrt{\frac{\sum_{i=1}^{R} \left((\sum_{j=1}^{C} key_j T_{ij}) - Mean \right)^2}{n}} \\ \text{s.t.} \quad & \textstyle\sum_{i=1}^{R} \mathbf{T}_{ij} = 1, \text{for all } j = (1, .., C) \\ & \mathbf{T}_{ij} = 0 \; or \; 1, \text{for all } i = (1, .., R) \\ & and \; j = (1, .., C) \end{aligned} \tag{6.4}$$

The distribution algorithm is shown in Alg.9. Data clusters assignment algorithm assigns C clusters to R partitions, one partition for one reducer. Every reducer pulls its data from its corresponding partition to achieve the reduce task. Lines 1-5 in Alg.9 clear the array called re-

Algorithm 9: Data clusters assignment algorithm-homogeneous

Input: Input: A collection of **R** reducers.
C data clusters to be distributed, which are numbered $Cluster_1$ through $Cluster_C$.
Number of keys in every data cluster i is $Key[i]$ where $(i = 1, 2, ..., \mathbf{C})$
Result: $T[1,2,..R][1,2,..,C]$

```
1  begin
2      for i = 1 to R do
3          Reducer_load [i] = 0;
4          T[i][j] = 0; for all j = 1 to C
5      end
6      Sorted_key[] = sort_descending(Key[]);
7      Index_Sorted_key[]=index of Sorted_key[ ] in Key[] ;
8      for i = 1 to C do
9          j = Index_Sorted_key[i];
10         if (keys of Cluster_j are the Map output of Mapper/Reducer node) && (this
             node produces more than half of data cluster j) then
11             z = Index(Mapper/Reducer node);
12         else
13             z = Minimum(Reducer_load[1,..,R]);
14         end
15         T[z][j] = 1;
16         Reducer_load[z] = Reducer_load[z] + Key[j];
17     end
18     Return  T [1,2,..R][1,2,..,C]
19 end
```

ducer_load, it stores the current load on the reducer. At the beginning of the distribution the value of *reducer_load* of every reducer equals 0. The algorithm creates $R \times C$ array to store the main assignment table as shown in Fig. 6.5. At the beginning of the algorithm, all values of main assignment table equal 0. During the process of the algorithm, if a cell inside this table has been assigned value of 1, then data cluster of corresponding column number is assigned to the reducer of corresponding row number in the main assignment table.

It is important to sort the data clusters based on their size in descending order. The reason of this descending order is to start the assignment of the largest data clusters first then the smaller size. Leaving the small sizes data clusters not distributed until the end of distribution makes it

easier to balance the load among reducers. In lines $6-7$, the function *Sort_descending* sorts the sizes of data clusters, *key*[], in descending order. The results are saved in *sorted_key*[] array and their original index in *Key*[] are stored in array named *Index_sorted_key*[]. For example: $Key = [22, 41, 11, 32]$, then $sorted\_key = [41, 32, 22, 11]$, and $Index\_sorted\_key = [2, 4, 1, 3]$. Line 9 takes the next data cluster from *sorted_key*[], and store its index on the original data clusters sequence.

Lines $10-11$ in Alg. 9 is the part that minimizes the data transfer over the network. A DataNode may be selected to execute both map and reduce tasks of a job, we call it Mapper/Reducer node. BDCP takes advantage of this opportunity to reduce the network traffic during the shuffle phase. The amount of data transferring over the network can be minimized by keeping the output of map task on a DataNodeas as an input of reduce task on the same node. Assume a DataNode X acts as Mapper/Reducer node for specific MapReduce job. In the proposed algorithm, if DataNode X produces more than half of the size of any data cluster during map task, the partitioning algorithm assigns this data cluster to DataNode X for reduce task. Line 13 assigns the data cluster to the the minimum load reducer for the current iteration. The data load of the selected reducer is updated in line 16 by adding the size of the data cluster to its load. BDCP produces the Main Assignment Table (MAT) as shown in Fig.6.5. MAT is the partitioner that used during the shuffle phase to map every key to its dedicated reducer. The final values of T_{ij} determine the reducer of every data cluster.

During the assignment process, in every iteration, line 15 assigns value of 1 to the selected row and column in the main assignment table. The time complexity of Alg. 9 can be analyzed as follows: each of line 3 and line 4 takes O(1) time. The time of the loop in lines 2–5 depends on R, where R is the number of reducers. Therefore, the time complexity of this loop is O(R). Line 6 is a descending order for the sizes of clusters, the worst case of the time complexity is:O(C^2). The time of the loop in lines 8–17 depends on C, where C is the number of data clusters to be distributed. Therefore, the time complexity of this loop is O(C). So, the total time complexity of data

assignment algorithm is O (N) + O(C^2). The partitioning algorithm is different in heterogeneous cluster environments. Since the reducers are usually belonging to different hardware generations, reducers may differ in their processing capability. Name node receives, from every reducer, the reducer data processing rate through the heart beats. Sampling phase is the perfect time for checking the current data processing speed of reducers before the starting of actual *MapReduce* job. *BDCP* assigns a processing ratio to the reducers and calculate the ratio of data every reducer should get to satisfy the balancing capacity.

	Key_1	Key_2	Key_3	Key_4	.....	Key_C
$Reducer_1$	1	0	0	1	.....	0
$Reducer_2$	0	1	0	0	.....	0
$Reducer_3$	0	0	0	0	.....	1
⋮			⋮			
$Reducer_R$	0	0	1	0	.....	0

Figure 6.5: Main assignment table

During the sampling phase, *BDCP* calculates, d_i, the amount of data processed between two successive heart beats by reducer i. So, the processing ratio, p_i, of reducer i can be calculated by *BDCP* during the sampling phase as shown in Eq. 6.5.

$$p_i = \frac{d_i}{\sum_{j=1}^{R} d_j} \tag{6.5}$$

The value of p_i is the approximate ratio that reducer i should get from the total intermediate data, where $(i = 1, 2, ...R)$. The size of data that should be assigned to reducer i, noted as sh_i, is calculated using Eq. 6.6

$$sh_i = p_i \times D \tag{6.6}$$

D is the total estimated data size of intermediate data. During data clusters distribution,

every time a data cluster is assigned to partition, the share, sh_i of the reducer corresponding to this partition is decreased. The partitioning algorithm in heterogeneous reducers is designed as a modified multiple knapsack problem, where every knapsack has a capacity, we called it balancing capacity. The load distributed evenly across knapsacks if every knapsack maintains a data load within its balancing capacity. Every reducer is considered as a knapsack, the balancing capacity of reducer i is, sh_i, as shown in Eq. (6.6). Let's assume there are C data clusters, the frequency of keys in $cluster_j$ is key_j, where $(j = 1,2,..,C)$. These data clusters have to be partitioned into R reducers, the data share of reducer i that satisfy the balancing capacity is sh_i, then the balancing partitioning algorithm can be calculated based on Eq. 6.7.

$$\begin{aligned} \min_{T_{ij}} \quad & \sum_{i=1}^{R} \left| \left(\sum_{j=1}^{C} \left(key_j \times T_{ij} \right) \right) - sh_i \right| \\ \text{s.t.} \quad & \sum_{i=1}^{R} \mathbf{T}_{ij} = 1, \text{for all } j = \left(1,..,C\right) \\ & \mathbf{T}_{ij} = 0 \; or \; 1, \text{for all } i = (1,..,R) \\ & and \; j = \left(1,..,C\right) \end{aligned} \tag{6.7}$$

Since the data clusters vary in their sizes, and one data cluster cannot be split into two reducers, then the partitioner cannot guarantee that the reducer receives data load that equals to its data share. However, the data load on a reducer is either larger than its share or lower than it. The partitioning algorithm minimizes this difference as much as possible in order to give better data load balancing among the reducers. As shown in the above minimizing equation, the balancing partitioner minimizes the absolute difference between the load on the reducer and the balancing capacity of the reducer.

In order to get better balanced distribution, the algorithm sorts the data clusters in descend-

ing order according to the number of keys in each data cluster. The algorithm selects the first data cluster in the sorted array (data cluster with largest size), assigns it to the reducer with the largest balancing capacity , and update the remaining capacity of the reducer. In the second round, the algorithm selects the second data cluster in the array (second largest size data cluster), assigns it to the biggest capacity reducer, updates the remaining capacity of the reducer, and so on until it reaches to the last data cluster (smallest data cluster). However, with the progress of the distribution algorithm, the sizes of data clusters become smaller, and the capacity of reducers gets lower.

The reason of leaving the smaller data clusters to the end of distribution is because the small size data clusters are easier to be distributed without effecting the overall balancing of data load among reducers. The data clusters assignment algorithm for heterogeneous reducers is shown in Alg.10. Lines $2-4$ initiate the reducers competition array. The initial capacity of every reducer is the data share of reducer calculated in Eq. (6.6). The value that represents capacity of reducer in the competition array is decreased every time the reducer has been assigned a data cluster. Line 5 sorts the sizes of data clusters in descending order as mentioned earlier in the Alg. 9.

In line 9, for every data cluster, the algorithm checks if the mapper node that produce the keys of this data cluster is Mapper/Reducer node. When such a case exists, and the Mapper/Reducer node produces more than half of the size of this data cluster, during the map phase, the data cluster is assigned to the Mapper/Reducer node for reduce task. Line 12 is the normal mode of distribution. The function $elected\_R()$ returns the index of reducer with the minimum data load. Line 14 is for assigning the data cluster to the selected reducer. Line 15 updates the capacity of the selected reducer, and so no until the algorithm reaches the last data cluster in the array (smallest data cluster).

In both homogeneous and heterogeneous reducers environments, during the sampling phase, the selected sample size controls the accuracy of representation of input data in the sample. If the sample size is small, many keys in the input data may not appear in the sample. The smaller the sample size, the higher probability the key is not appearing in the sampling phase.

Algorithm 10: Data clusters assignment algorithm-heterogeneous

Input: Input: A collection of **R** reducers.
$\mathbf{sh_i}$ The share of the intermediate data of $reducer_i$ that satisfying balancing capacity. Calculated using Eq. (6.6)
C data clusters to be distributed, which are numbered $Cluster_1$ through $Cluster_C$.
Key$[i]$ is the number of keys in data $Cluster_i$
Result: $T[1,2,..R][1,2,..,C]$

```
1  begin
2      for i = 1 to R do
3          reducer_compete [i] = sh[i];
4      end
5      Sorted_key[] = sort_descending(Key[]);
6      Index_Sorted_key[]=Index of original order ofsort_key[];
7      for i = 1 to C do
8          j = Index_Sorted_key[i];
9          if (keys of Cluster_j are the Map output of Mapper/Reducer node) && (this
             node produces more than half of data cluster j) then
10             z = Index(Mapper/Reducer node);
11         else
12             z = elected_R (reducer_compete[1,2,..R])
13         end
14         T[z][j] = 1;
15         reducer_compete[z] = reducer_compete[z] − Key[j];
16     end
17     Return T[1,2,..R][1,2,..,C]
18 end
```

It is very normal situation when many keys are not presented in the sample. *BDCP* is designed for partitioning the keys that appear in the sampling phase. Moreover, it calculates the actual size of data based on the resulted data sample. To solve this issue for those keys that do not appear in the sample, *BDCP* applies the default HashPartitioner on those data clusters because their size is very small in the actual input data, and do not cause reducers data skew. The partitioning process using *BDCP* adds extra phase, (sampling phase), to *MapReduce* job phases. The sampling phase cannot be overlapped with the other phases. Sampling phase makes the actual map tasks on input data starts later than the actual job start time. This delay results in minimizing the reduce phase time, and slightly decreasing the shuffle phase time. As illustrated in Fig. 6.6.

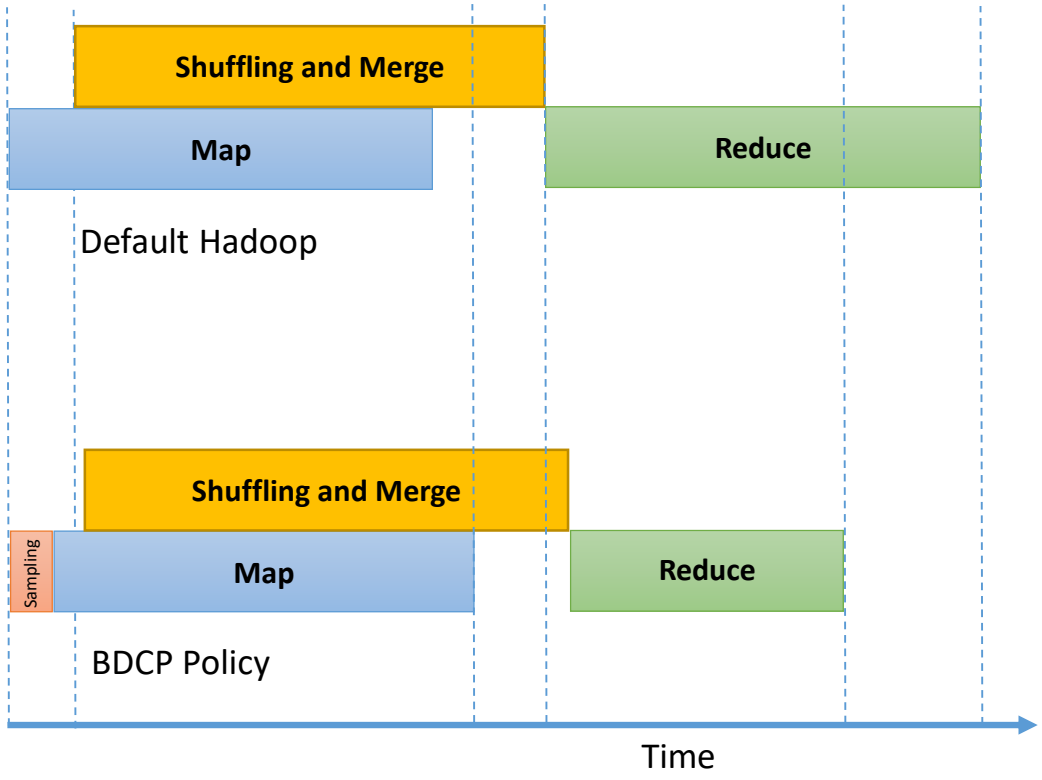

Figure 6.6: MapReduce phases for default Hadoop and BDCP

6.4 Evaluation

To test the performance of the proposed strategy, homogeneous and heterogeneous Hadoop cluster environment have been prepared. Practical MapReduce application jobs were chosen to evaluate the performance of *BDCP*. The crucial factor is to minimize the job execution time. The performance of the suggested algorithm is examined by comparing the execution time of reduce phase using this algorithm against other algorithms. Two different types of benchmarks are used for synthetic and real-world datasets with different data skew rate are used to evaluate *BDCP* in homogeneous and heterogeneous Hadoop clusters. We compared *BDCP* with the default Hadoop HashPartitioner, and Range partitioner [3] in same experiments environment. Hadoop HashPar-

titioner is the default mechanism in Hadoop environment which can obtain a good performance only when the $(Key, Value)$ pairs are distributed uniformly. Range partitioner is a widely used algorithm of partition distribution in which the intermediate $(Key, Value)$ pairs are sorted by key first, and then the pairs are assigned to reduce tasks according to this key range sequentially. Range is one of the algorithms that can improve the data balance among reduce tasks.

The performance of default Hadoop hash-partitioner, range partitioner, and BDCP have been compared based on the reduce phase execution time in order to verify the effect of intermediate data placement. Heavy MapReduce jobs that process large amounts of input data, and generate large intermediate data are implemented. In order to ensure accuracy, each group of experiments has been executed at least 10 times, and the mean value has been used as result. the proposed system is implemented on Hadoop YARN version 2.9.2. During the sampling phase, 10% of the input data has been chosen as the sample size. The experiments have been implemented on homogeneous cluster environment, then the same experiments on implemented on heterogeneous cluster environment.

6.4.1 Homogeneous cluster experiments

The experiments are conducted on a Hadoop YARN cluster consists of 20 physical machines connected on single switch with 1 Gbps network bandwidth, installed with Ubuntu 14.10. with 8 cores 2.53 GHz processors, 16G memory, 1TB hard disk. The proposed system has been implemented and evaluated by running different types of benchmarks.

6.4.1.1 Word count benchmark testing

Three algorithms are compared under Word Count benchmark. Word Count job counts the number of each key in a file and produces an output file containing all keys and their frequencies. a heavy MapReduce job has been used to processes large amount of input data. The input data is split into blocks in HDFS. Each block is processed line by line to count the number of keys. Word Count

job is suitable job to test the proposed algorithm because it generates large intermediate data [55].

Fig. 6.7 shows the reduce phase execution time of word count job on 6 GB of data file, the skew degree of the data used ranges from 0.1 to 1.1. As shown in the figuer, as the data skew increases the processing time gets larger because of the data skew leads to reducers skew especially with using the HashPartitioner. At the beginning, the execution times is relatively low. The increasing of data skew has big impact on the reduce phase execution time using Hadoop HashPartitioner and lower impact on Range algorithm, while BDCP algorithm mitigates this impact and shows slightly increasing in processing time as the data skew increases.

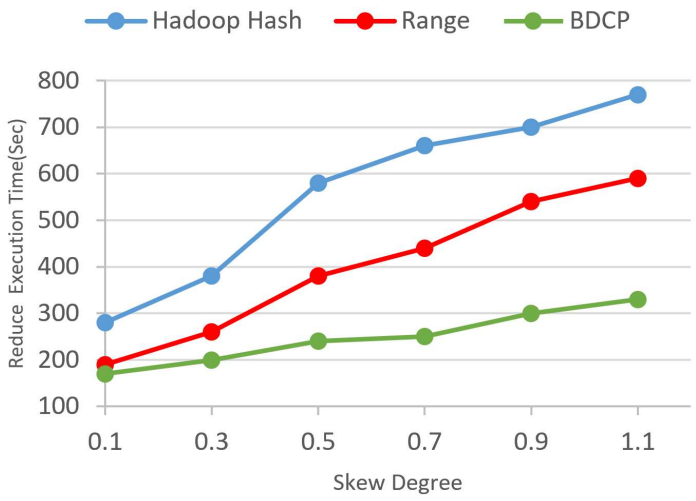

Figure 6.7: Word count job on a file of size 6 GB, and 0.1 to 1.1 skew degree in homogeneous cluster

Fig.6.8 shows the reduce execution time of word count job on files with data sizes, 2 GB, 4 GB, and 6 GB. The skew degree of the data used is 0.1. Even though the data skew is low, the increasing of file size makes the reduce phase execution time increases with different ratios depending on the used partitioner. Fig.6.9 shows the reduce execution time of word count job on file with data sizes, 2 GB, 4 GB, and 6 GB and the skew degree of the data used is 1.1. Because the data skew is high, the reduce phase execution time is high even when the file size is not large.

The increasing of file size makes the execution time longer, because the bigger data file the more intermediate data skew for the same input data skew degree.

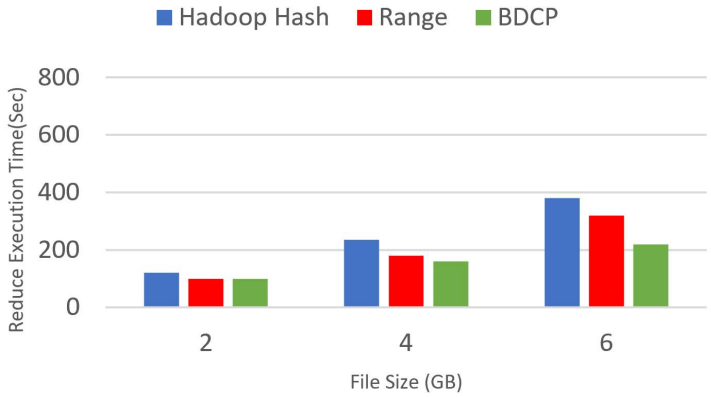

Figure 6.8: Word count job on files of sizes, 2, 4, and 6 GB. skew degree is 0.1. in homogeneous cluster

6.4.1.2 Sort benchmark test

Sorting is a part of widely adopted benchmarks for parallel computing [39].Sort job, a reduce-input- heavy job, is used to test the proposed algorithm by processing input data with different data skew degrees. sort benchmark job has been implemented on file with size 6 GB of data, and the skew degree of the data used ranges from 0.1 to 1.1.

The reduce phase execution time of BDCP is shorter than Hadoop HashPartitioner and Range when processing the data with high skew rate. if the data skew rate is lower than a certain value, BDCP performance is better but it is close to the performance of the other two algorithms. While BDCP performs much better with the increase of data skew. As shown in Fig. 6.10, when data skew degree is less than 0.40, the Hadoop HashPartitioner has an execution time closer to the other two algorithms because of its even partitions of intermediate data.

When the skew becomes more than 0.3 BDCP starts to outperform the Hadoop HashPari-

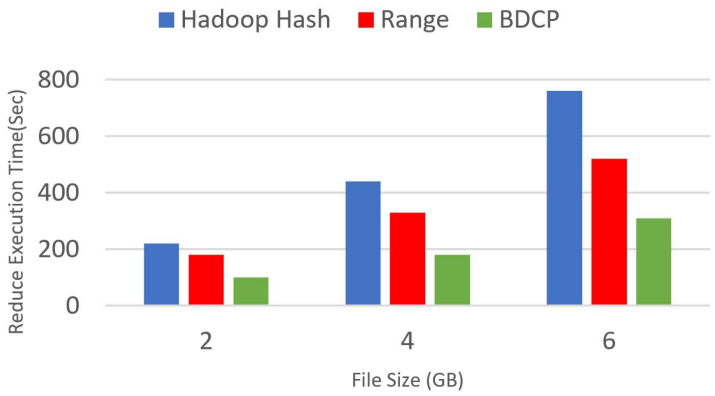

Figure 6.9: Word count job on files of sizes, 2, 4, and 6 GB. skew degree is 1.1. in homogeneous cluster

tioner and Range algorithms. When both the data skew degree and file size are small, the reduce phase execution times for the three algorithms are relatively low. However, the increasing of file size with the same data skew degree makes the execution time slightly higher. However, *BDCP* has lower execution time than other two algorithms with the increase of the file size. Fig. 6.11 shows the reduce phase execution time of Sort jobs on files with data sizes, 2 GB, 4 GB, and 6 GB, the skew degree of the data used is 0.1.

Using higher data skew for different sizes of data files shows that BDCP highly outperforms both algorithms. With the increasing of data file for the same (high) data skew, the execution time of reduce phase of BDCP becomes less than half of the reduce phase execution time using HashPartitioner, and less than 0.6 of the reduce phase execution time using Range partitioner. Fig.6.12 shows the reduce execution time of Sort jobs on files with sizes, 2 GB, 4 GB, and 6 GB of data, the skew degree of the data used is 1.1.

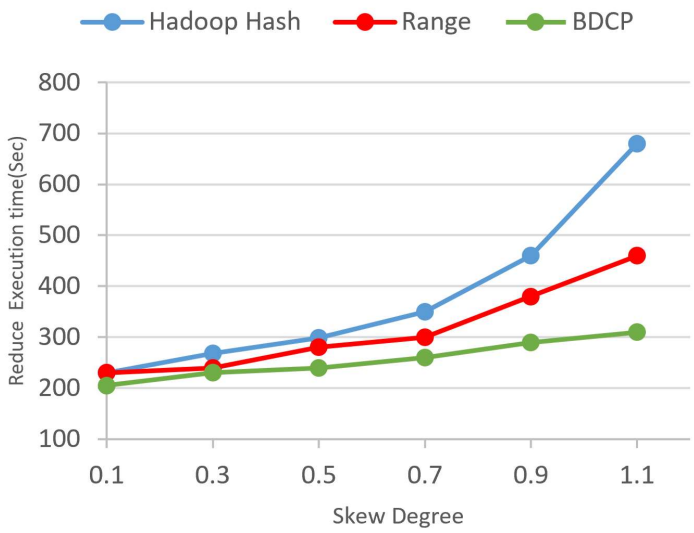

Figure 6.10: Sort job on a file of size 6 GB, and 0.1 to 1.1 skew degree in homogeneous cluster

6.4.2 Heterogeneous cluster experiments

In order to implement the partitioning algorithm in a heterogeneous cluster environment where the reducers vary in their available resources, experiments are conducted on a Hadoop YARN cluster consists of 20 physical machines connected on single switch with network bandwidth of 1 Gbps installed with Ubuntu 14.10, as following:

- 10 machines, 8 core 2.53 GHz processors, 16G memory.
- 5 machines with 4 core 3.4 GHz processor, 8G memory.
- 5 machines with 4 core 2.7 GHz processors, 4G memory.

The proposed system has been implemented and evaluated by running different types of benchmarks. The data skew and the variation in DataNodes processing capabilities are the main two reasons that cause straggler tasks.

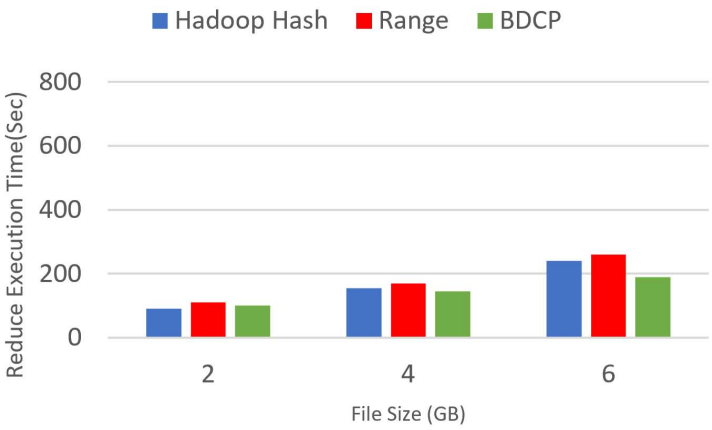

Figure 6.11: Sort job on files of sizes, 2, 4, and 6 GB. Skew degree is 0.1.in homogeneous cluster

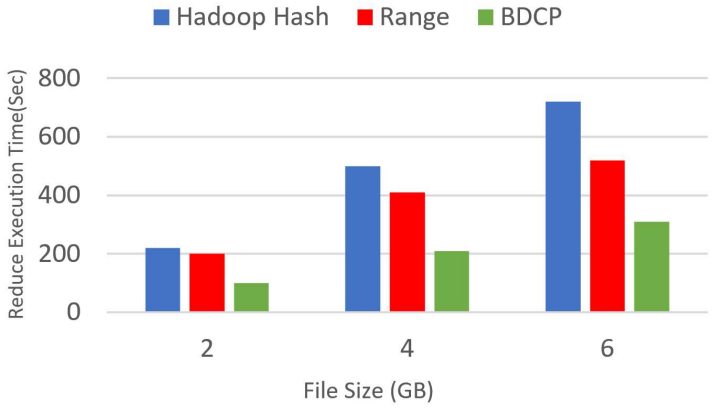

Figure 6.12: Sort job on files of sizes, 2, 4, and 6 GB. Skew degree is 1.1.in homogeneous cluster

In this experimental environment these two factors are exist. In this experiment the impact of these factors on the reduce phase running time is examined by running different types of benchmarks.

6.4.2.1 Word count benchmark

Heterogeneous Hadoop cluster with the mentioned configuration produces unbalanced distribution of intermediate data load among reducers nodes which results in increasing the processing time of reduce phase especially in HashPartitioner. However the increasing of data skew Aggravates the problem of balancing the data load among reducer nodes. Fig. 6.13 shows the reduce execution time of word count jobs on a files with 6 GB of data, the skew degree of the data used ranges from 0.1 to 1.1. The figure shows that the reduce phase processing time in BDCP is lower than the reduce phase processing time of the other two algorithms for all skew degrees.

Even though the processing times for the three algorithms are relatively low when the the data skew is low, they are much higher than those of same job in homogeneous cluster environment. The reduce phase processing time increases with the increasing of data skew with the existing of variation of reducers processing capabilities. BDCP shows good mitigation for the increasing of data skew in heterogeneous cluster environment.

Fig.6.14 shows the reduce phase execution time of word count job with data sizes, 2 GB, 4 GB, and 6 GB, the skew degree of the data used is 0.1. As shown the Figs. 6.13 and 6.14, even though when the data skew is low, BDCP is achieving better than Hadoop HashPartitioner and Range because it considers the variation of computing capabilities of the reducers. While HashPartitioner, and range took longer time than the time they took previously in homogeneous cluster environment. Fig. 6.15 shows the reduce execution time of word count job on a file with data sizes, 2 GB, 4 GB, and 6 GB, on a heterogeneous Hadoop cluster, the skew degree of the data used is 1.1.

Both high data skew and variation in computing capabilities of the reducers result in higher reduce execution time for all algorithms. The results show that BDCP algorithm always has the shortest reduce time for different file sizes with high skew degrees. However the increasing of file size, for the same input data skew degree, increases the execution time of reduce phase.

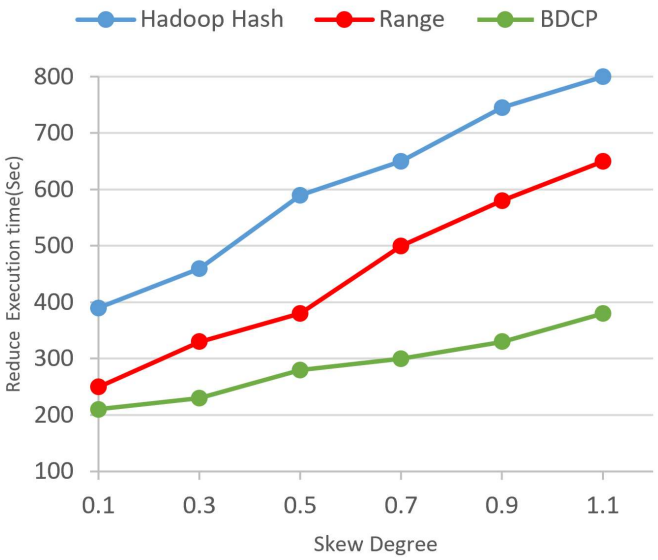

Figure 6.13: Word count job on a file of size 6 GB, and 0.1 to 1.1 skew degree in heterogeneous cluster

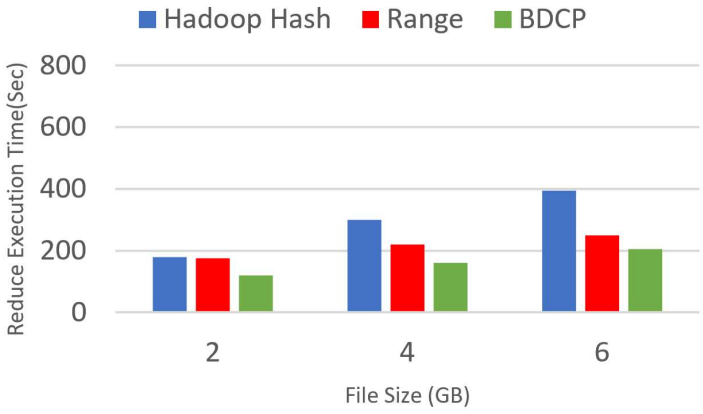

Figure 6.14: Word count job on files of sizes, 2, 4, and 6 GB. Skew degree is 0.1.in heterogeneous cluster

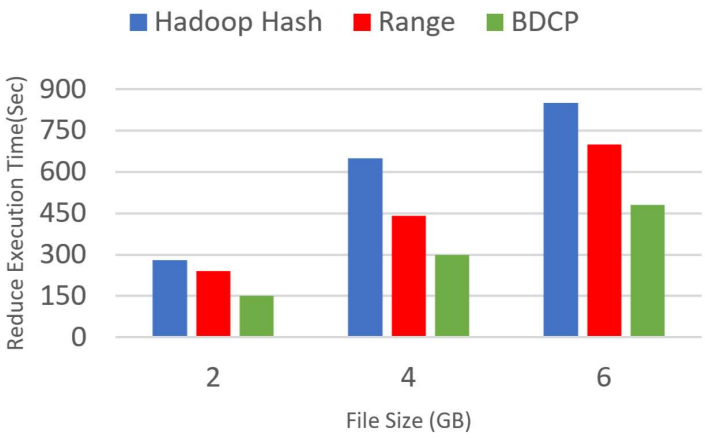

Figure 6.15: Word count job on files of sizes, 2, 4, and 6 GB. Skew degree is 1.1.in heterogeneous cluster

6.4.2.2 Sort benchmark test

Sort benchmark job has been implemented on a file with 6 GB of data, the skew degree of the data used ranges from 0.1 to 1.1. Fig. 6.16 shows the reduce execution time of the sort job. BDCP works in similar efficiency to Hadoop HashPartitioner and Range when the data skew rate is lower than 0.2, but it gives shorter execution time, while it performs better with the increasing of data skew. BDCP is much faster than Hadoop HashPartitioner and Range in processing the data with high skew rate.

Hadoop HashPartitioner of intermediate data does not consider the heterogeneity on the reducers, so it takes more time than the previous sort experiment on homogeneous environment. Fig.6.17 illustrates the reduce phase execution time of Sort job of file with sizes, 2 GB, 4 GB, and 6 GB, the skew degree of the data used is 0.1. BDCP performs better than HashPartitioner and Range. The differences in reduce phase execution times for the three algorithms increase with the increasing of the file size for the same skew degree. However, the variations in reduce phase execution time are not large because the skew degree is low.

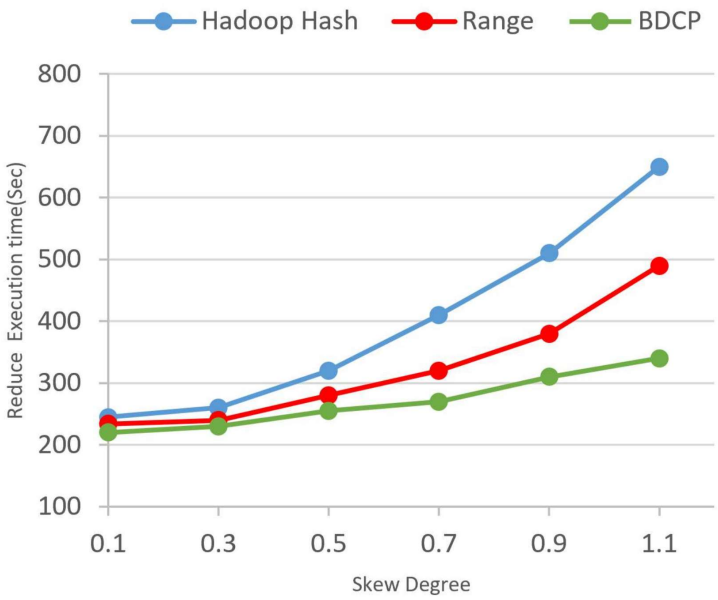

Figure 6.16: Sort job on a file of size 6 GB, and 0.1 to 1.1 skew degree in heterogeneous cluster

Fig. 6.18 shows the reduce phase execution time of Sort jobs on files with data sizes, 2 GB, 4 GB, and 6 GB, the skew degree of the data used is 1.1. For this high skew degree, the reduce phase execution times for the three algorithms are relatively high and increase with the increasing of the file size. However, the variations in reduce phase execution time are large because the skew degree is high. BDCP outperforms HashPartitioner and range especially with the increasing of file size with high data skew.

To illustrate the timing of all the MapReduce job phases, a Word Count job on a file with size of 2 GB of data and the skew degree is 0.3 has been implemented using the same configuration of heterogeneous cluster environment. BDCP algorithm divides MapReduce job execution process into five phases, as shown in Fig. 6.19. The phases are represented on the figure are sampling, map, overlapped map and shuffle, shuffle, and reduce phase. Sampling phase is only used by BDCP algorithm. It can not be overlapped with map phase. In BDCP, map phase starts right after sampling phase.

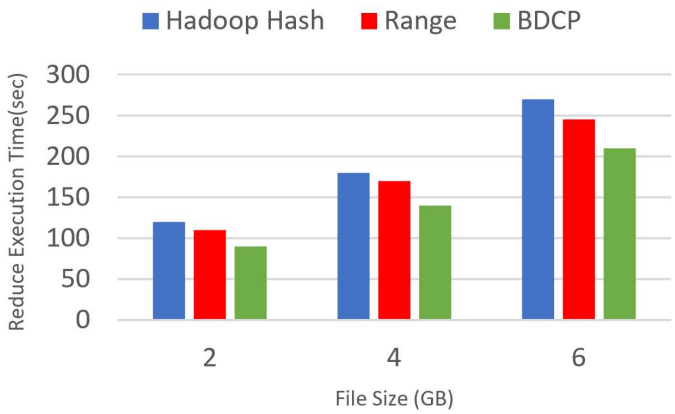

Figure 6.17: Sort job on files of sizes, 2, 4, and 6 GB. Skew degree is 0.1. in heterogeneous cluster

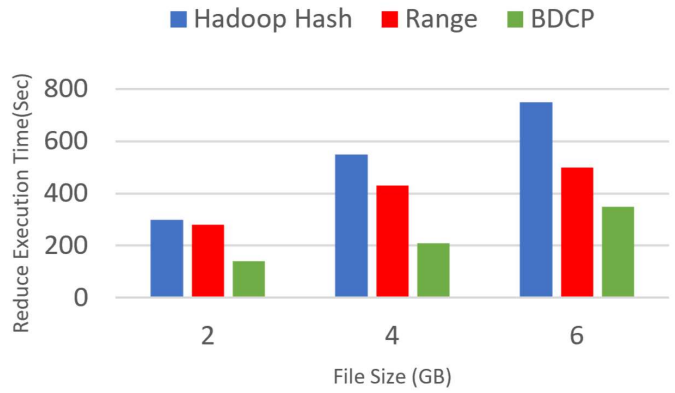

Figure 6.18: Sort job on files of sizes, 2, 4, and 6 GB. Skew degree is 1.1. in heterogeneous cluster

Because the partitioning policy has been already generated after the sampling phase, shuffle phase starts whenever there is an output from the map phase. While the Hadoop HashPartitioner begins shuffling the map task outputs when 5% of map tasks have completed. The execution time of reduce phase is minimized in BDCP as shown in Fig.6.19.

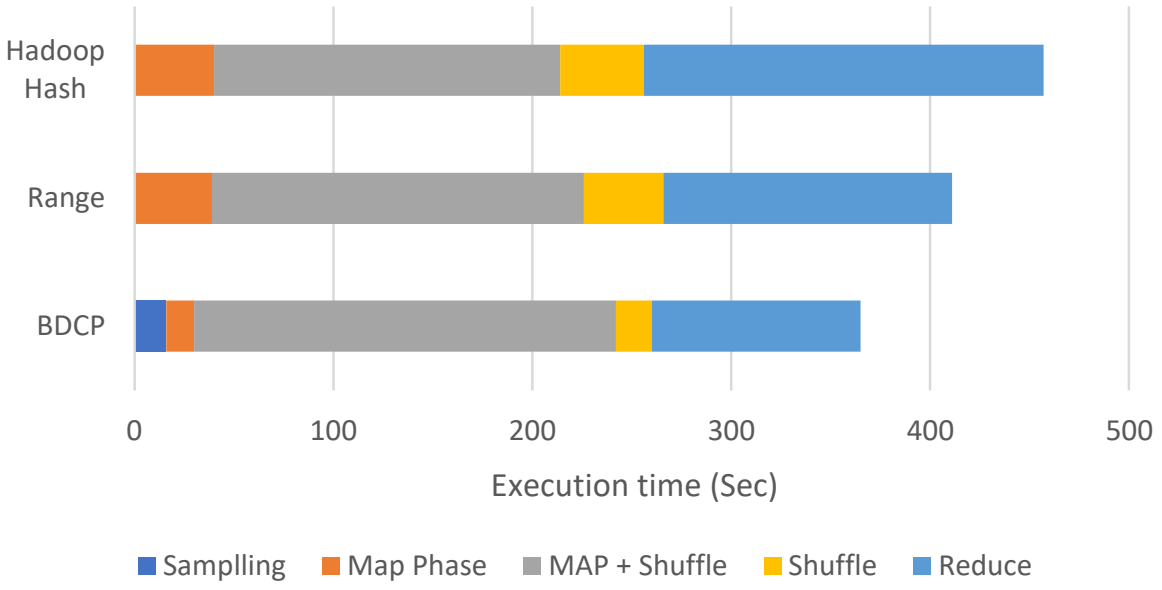

Figure 6.19: MapReduce phases of Word Count job on a file of size 2 GB, and skew degree is 0.3.

CHAPTER 7: CONCLUSION AND PROPOSED FUTURE WORK

This dissertation proposes four algorithms to improve the performance of data-intensive applications running on cloud platforms from three different aspects: data load distribution across cluster nodes, map task straggler identification, and reduce task straggler mitigation. The net effect of the proposed algorithms is to improve the efficiency of task execution and minimize the job completion time. In this dissertation a new replica placement policy for HDFS, named Intelligent Data Placement Mechanism (IDPM), has been proposed. The issue of data load balancing among cluster nodes is addressed in this policy by evenly distributing replicas to cluster nodes. By using IDPM, there is no more need for the load balancing utility used by HDFS because IDPM generates perfectly even replica distribution and satisfies all HDFS replica placement rules as confirmed by the experimental results. IDPM is designed for cluster environments in which all cluster nodes are similar in their computing capabilities.

However, in heterogeneous cluster environments, an even replica distribution does not mean it is a fair distribution. The fair distribution can be achieved only when the number of replicas that are assigned to a node is based of the available resources for this node. If every file in HDFS is fairly distributed to the DataNodes then the cluster load balancing and the cluster efficient utilization is satisfied. In this Dissertation, a new replica placement policy for HDFS has been proposed for heterogeneous cluster environments that have variations of hardware computing capabilities. The proposed policy, named Speed-based Replica Assignment Policy SRAP, can remarkably improve data load balancing among heterogeneous Hadoop cluster. SRAP can generate replicas distribution in a way where all cluster nodes are fully utilized, and data processing throughput is increased, as confirmed by the evaluation results. There is an exciting future work for the proposed policy, SRAP can be developed to be used for environments where hardware with more sophisticated heterogeneity is present in cloud networks and the amount of memory of DataNode is added as a factor beside the processing capability of the DataNode.

The drawbacks of the commonly used straggler detection and mitigation methods are analyzed in this dissertation. The identification of straggler task and the time of its identification during the progress of running task is very critical part in straggler mitigation process. The earlier the straggler tasks are identified the more successful buck up tasks are initiated. Progress and Feedback based Speculative Execution (PFSE) algorithm is proposed in this dissertation as an improved straggler identification scheme. PFSE uses phase level progress and feedback task information to estimate task completion time in order to identify straggler task as early as possible. Extensive experiments have been conducted to evaluate the performance of PFSE compared with another previously suggested algorithms, LATE and SLM. The results indicate that PFSE gives better accuracy in the estimation of task completion time, eventually it produces higher successful backup task rate. Future work can be proposed to use PFSE to build an algorithm to estimate the task remaining execution time in the other MapReduce phases.

In the reduce phase of MapReduce jobs, the straggler reduce tasks mainly results from the unbalanced distribution of the intermediate data to the reducers. Unlike the map phase, distribution of data to the reducers can not be decided before the start of the reduce phase because the distribution of the intermediate data is different for every data set and map task. Balanced Data Clusters Partitioner (BDCP) algorithm is proposed to minimize the effect of the commonly known problem of intermediate data skew, and mitigate the straggler reduce task by balancing the distribution of intermediate data to the reducers. BDCP algorithm adds a new phase to MapReduce job phases, it is the sampling phase. An estimation of the intermediate data distribution is calculated, and feedback about reducer processing capabilities is received during the sampling phase in order to create the balanced partitioning policy. Extensive experiments have been implemented on different data sizes with different skew degrees to evaluate the performance of BDCP compared with Hadoop HashPartitioner and Range algorithms. To evaluate the performance of BDCP in hadoop computing environments, the proposed system is evaluated by running different types of benchmarks in homogeneous and heterogeneous Hadoop clusters. Word Count benchmark test and Sort bench-

mark test are implemented to evaluate the reduce phase execution time in BDCP. The extensive experiments show that the MapReduce job completion time when using the partitioner BDCP is shorter than the job completion time of the same job when the HashPartitioner of Hadoop or Range partitioner is used. Future research line can be suggested to add the network bandwidth between cluster nodes as a factor in the partitioning algorithm of intermediate data.

APPENDIX : PERMISSION TO REUSE PUBLISHED MATERIAL

Title: Intelligent Data Placement Mechanism for Replicas Distribution in Cloud Storage Systems

Conference Proceedings: 2016 IEEE International Conference on Smart Cloud (SmartCloud)

Author: Ibrahim Adel Ibrahim; Wei Dai; Mustafa Bassiouni

Publisher: IEEE

Date: 18-20 Nov. 2016

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Title: Improving MapReduce Performance with Progress and Feedback Based Speculative Execution

Conference Proceedings: 2017 IEEE International Conference on Smart Cloud (SmartCloud)

Author: Ibrahim Adel Ibrahim

Publisher: IEEE

Date: Nov. 2017

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

LIST OF REFERENCES

[1] Mapreduce job. word count. `http://spark.apache.org/examples.html`. Accessed: 27.04.2019.

[2] Mapreduce: Official apache hadoop website. `http://hadoop.apache.org`. Accessed: 14.02.2019.

[3] Range partitioner, [eb/ol]. `http://spark.apache.org/docs/1.3.0/api/java/org/apache/spark/RangePartitioner.html`. Accessed: 11.04.2019.

[4] M. Alam and K. A. Shakil. Recent developments in cloud based systems: state of art. *arXiv preprint arXiv:1501.01323*, 2015.

[5] E. Amazon. Amazon elastic compute cloud. *Retrieved Feb*, 10, 2009.

[6] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 185–198, 2013.

[7] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Osdi*, volume 10, page 24, 2010.

[8] R. Chansler, H. Kuang, S. Radia, K. Shvachko, and S. Srinivas. The architecture of open source applications: the hadoop distributed file system. *Retrieved from URL: https://web. archive. org/web/20131110025205/http://www. aosabook. org/en/hdfs. html on*, pages 1–12, 2017.

[9] H.-L. Chen and S.-H. Chen. A balanced partitioning mechanism using collapsed-condensed trie in mapreduce. In *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, pages 97–102. IEEE, 2018.

[10] H.-L. Chen and Y.-S. Shen. Reducing imbalance ratio in mapreduce. In *2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2)*, pages 279–282. IEEE,

2017.

[11] L. Chen, W. Lu, L. Wang, E. Bao, W. Xing, Y. Yang, and V. Yuan. Optimizing mapreduce partitioner using naive bayes classifier. In *2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*, pages 812–819. IEEE, 2017.

[12] Q. Chen, C. Liu, and Z. Xiao. Improving mapreduce performance using smart speculative execution strategy. *IEEE Transactions on Computers*, 63(4):954–967, 2014.

[13] Q. Chen, J. Yao, and Z. Xiao. Libra: Lightweight data skew mitigation in mapreduce. *IEEE Transactions on parallel and distributed systems*, 26(9):2520–2533, 2015.

[14] Q. Chen, D. Zhang, M. Guo, Q. Deng, and S. Guo. Samr: A self-adaptive mapreduce scheduling algorithm in heterogeneous environment. In *2010 10th IEEE International Conference on Computer and Information Technology*, pages 2736–2743. IEEE, 2010.

[15] Y. Chen, Z. Liu, T. Wang, and L. Wang. Load balancing in mapreduce based on data locality. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 229–241. Springer, 2014.

[16] D. Cheng, J. Rao, Y. Guo, C. Jiang, and X. Zhou. Improving performance of heterogeneous mapreduce clusters with adaptive task tuning. *IEEE Transactions on Parallel and Distributed Systems*, 28(3):774–786, 2016.

[17] Z. Cheng, Z. Luan, Y. Meng, Y. Xu, D. Qian, A. Roy, N. Zhang, and G. Guan. Erms: An elastic replication management system for hdfs. In *2012 IEEE International Conference on Cluster Computing Workshops*, pages 32–40. IEEE, 2012.

[18] W. Dai and M. Bassiouni. An improved task assignment scheme for hadoop running in the clouds. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(1):23, 2013.

[19] W. Dai, I. Ibrahim, and M. Bassiouni. Improving load balance for data-intensive computing on cloud platforms. In *2016 IEEE International Conference on Smart Cloud (SmartCloud)*,

pages 140–145. IEEE, 2016.

[20] W. Dai, I. Ibrahim, and M. Bassiouni. A new replica placement policy for hadoop distributed file system. In *2016 IEEE 2nd International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing (HPSC), and IEEE International Conference on Intelligent Data and Security (IDS)*, pages 262–267. IEEE, 2016.

[21] J. Dhok and V. Varma. Using pattern classification for task assignment in mapreduce. In *Proc. ISEC*, 2010.

[22] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson. Cohadoop: flexible data placement and its exploitation in hadoop. *Proceedings of the VLDB Endowment*, 4(9):575–585, 2011.

[23] K. Gai and S. Li. Towards cloud computing: a literature review on cloud computing and its development trends. In *2012 Fourth International Conference on Multimedia Information Networking and Security*, pages 142–146. IEEE, 2012.

[24] K. Gai, M. Qiu, and H. Zhao. Cost-aware multimedia data allocation for heterogeneous memory using genetic algorithm in cloud computing. *IEEE transactions on cloud computing*, 2016.

[25] K. Gai and A. Steenkamp. A feasibility study of platform-as-a-service using cloud computing for a global service organization. *Journal of Information Systems Applied Research*, 7(3):28, 2014.

[26] Y. Guo, J. Rao, C. Jiang, and X. Zhou. Moving hadoop into the cloud with flexible slot management and speculative execution. *IEEE Transactions on Parallel and Distributed systems*, 28(3):798–812, 2016.

[27] M. A. H. Hassan, M. Bamha, and F. Loulergue. Handling data-skew effects in join operations using mapreduce. *Procedia Computer Science*, 29:145–158, 2014.

[28] D. Howley. Is microsoft's onedrive the best cloud storage service?

[29] X. Huang, L. Zhang, R. Li, L. Wan, and K. Li. Novel heuristic speculative execution strategies in heterogeneous distributed environments. *Computers & Electrical Engineering*, 50:166–179, 2016.

[30] I. A. Ibrahim and M. Bassiouni. Improving mapreduce performance with progress and feedback based speculative execution. In *2017 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 120–125. IEEE, 2017.

[31] I. A. Ibrahim and M. Bassiouni. Improvement of data throughput in data-intensive cloud computing applications. In *2019 IEEE Fifth International Conference on Big Data Computing Service and Applications (BigDataService)*, pages 49–54. IEEE, 2019.

[32] I. A. Ibrahim, W. Dai, and M. Bassiouni. Intelligent data placement mechanism for replicas distribution in cloud storage systems. In *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 134–139. IEEE, 2016.

[33] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi. Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 17–24. IEEE, 2010.

[34] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The performance of mapreduce: An in-depth study. *Proceedings of the VLDB Endowment*, 3(1-2):472–483, 2010.

[35] W. Jing, D. Tong, Y. Wang, J. Wang, Y. Liu, and P. Zhao. Mamr: High-performance mapreduce programming model for material cloud applications. *Computer Physics Communications*, 211:79–87, 2017.

[36] H. Jung and H. Nakazato. Dynamic scheduling for speculative execution to improve mapreduce performance in heterogeneous environment. In *2014 IEEE 34th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 119–124. IEEE, 2014.

[37] D. Karapiperis and V. S. Verykios. Load-balancing the distance computations in record linkage. *ACM SIGKDD Explorations Newsletter*, 17(1):1–7, 2015.

[38] O. Khan, R. C. Burns, J. S. Plank, W. Pierce, and C. Huang. Rethinking erasure codes for

cloud file systems: minimizing i/o for recovery and degraded reads. In *FAST*, page 20, 2012.

[39] Z. Khatami, S. Hong, J. Lee, S. Depner, H. Chafi, J. Ramanujam, and H. Kaiser. A load-balanced parallel and distributed sorting algorithm implemented with pgx. d. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1317–1324. IEEE, 2017.

[40] K. Kumar and Y.-H. Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, (4):51–56, 2010.

[41] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 25–36. ACM, 2012.

[42] C.-W. Lee, K.-Y. Hsieh, S.-Y. Hsieh, and H.-C. Hsiao. A dynamic data placement strategy for hadoop in heterogeneous environments. *Big Data Research*, 1:14–22, 2014.

[43] D. Lee, J.-S. Kim, and S. Maeng. Large-scale incremental processing with mapreduce. *Future Generation Computer Systems*, 36:66–79, 2014.

[44] J. Li, Y. Liu, J. Pan, P. Zhang, W. Chen, and L. Wang. Map-balance-reduce: an improved parallel programming model for load balancing of mapreduce. *Future Generation Computer Systems*, 2017.

[45] Y. Li, W. Dai, Z. Ming, and M. Qiu. Privacy protection for preventing data over-collection in smart city. *IEEE Transactions on Computers*, 65(5):1339–1350, 2016.

[46] Y. Li, Q. Yang, S. Lai, and B. Li. A new speculative execution algorithm based on c4. 5 decision tree for hadoop. In *International Conference of Young Computer Scientists, Engineers and Educators*, pages 284–291. Springer, 2015.

[47] C. Lin, W. Guo, and C. Lin. Self-learning mapreduce scheduler in multi-job environment. In *2013 International Conference on Cloud Computing and Big Data*, pages 610–612. IEEE, 2013.

[48] C.-Y. Lin and Y.-C. Lin. A load-balancing algorithm for hadoop distributed file system. In

2015 18th International Conference on Network-Based Information Systems, pages 173–179. IEEE, 2015.

[49] Q. Liu, W. Cai, J. Shen, Z. Fu, and N. Linge. A smart speculative execution strategy based on node classification for heterogeneous hadoop systems. In *2016 18th International Conference on Advanced Communication Technology (ICACT)*, pages 223–227. IEEE, 2016.

[50] Q. Liu, W. Cai, J. Shen, Z. Fu, X. Liu, and N. Linge. A speculative execution strategy based on node classification and hierarchy index mechanism for heterogeneous hadoop systems. In *2017 19th International Conference on Advanced Communication Technology (ICACT)*, pages 889–894. IEEE, 2017.

[51] X. Liu and Q. Liu. An optimized speculative execution strategy based on local data prediction in a heterogeneous hadoop environment. In *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, volume 2, pages 128–131. IEEE, 2017.

[52] S.-Q. Long, Y.-L. Zhao, and W. Chen. Morm: A multi-objective optimized replication management strategy for cloud storage cluster. *Journal of Systems Architecture*, 60(2):234–244, 2014.

[53] P. Nonava. *Hdfs blocks placement strategy*. PhD thesis, Master's thesis, University of Fribourg, 2014.

[54] M. Qiu, Z. Ming, J. Li, K. Gai, and Z. Zong. Phase-change memory optimization for green cloud with genetic algorithm. *IEEE Transactions on Computers*, 64(12):3528–3540, 2015.

[55] K. Rattanaopas and S. Kaewkeeree. Improving hadoop mapreduce performance with data compression: A study using wordcount job. In *2017 14th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, pages 564–567. IEEE, 2017.

[56] T. Shabeera and S. M. Kumar. Bandwidth-aware data placement scheme for hadoop. In *2013 IEEE Recent Advances in Intelligent Computational Systems (RAICS)*, pages 64–67. IEEE,

2013.

[57] K. Shvachko, H. Kuang, S. Radia, R. Chansler, et al. The hadoop distributed file system. In *MSST*, volume 10, pages 1–10, 2010.

[58] V. Subramanian, L. Wang, E.-J. Lee, and P. Chen. Rapid processing of synthetic seismograms using windows azure cloud. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 193–200. IEEE, 2010.

[59] X. Sun, C. He, and Y. Lu. Esamr: An enhanced self-adaptive mapreduce scheduling algorithm. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 148–155. IEEE, 2012.

[60] J. Tan, A. Chin, Z. Z. Hu, Y. Hu, S. Meng, X. Meng, and L. Zhang. Dynmr: Dynamic mapreduce with reducetask interleaving and maptask backfilling. In *Proceedings of the Ninth European Conference on Computer Systems*, page 2. ACM, 2014.

[61] S. Tang, B.-S. Lee, and B. He. Dynamicmr: A dynamic slot allocation optimization framework for mapreduce clusters. *IEEE Transactions on Cloud Computing*, 2(3):333–347, 2014.

[62] Z. Tang, L. Jiang, J. Zhou, K. Li, and K. Li. A self-adaptive scheduling algorithm for reduce start time. *Future Generation Computer Systems*, 43:51–60, 2015.

[63] Z. Tang, X. Zhang, K. Li, and K. Li. An intermediate data placement algorithm for load balancing in spark computing environment. *Future Generation Computer Systems*, 78:287–301, 2018.

[64] E. G. Ularu, F. C. Puican, A. Apostu, M. Velicanu, et al. Perspectives on big data and big data analytics. *Database Systems Journal*, 3(4):3–14, 2012.

[65] A. Verma, L. Cherkasova, and R. H. Campbell. Aria: automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 235–244. ACM, 2011.

[66] L. Vu and G. Alaghband. A load balancing parallel method for frequent pattern mining on multi-core cluster. In *Proceedings of the Symposium on High Performance Computing*, pages

49–58. Society for Computer Simulation International, 2015.

[67] Y. Wang, W. Lu, R. Lou, and B. Wei. Improving mapreduce performance with partial speculative execution. *Journal of grid computing*, 13(4):587–604, 2015.

[68] T. White. Hadoop: The definitive guide, chapter meet hadoop, 2015.

[69] H. Wu. Big data management the mass weather logs. In *International Conference on Smart Computing and Communication*, pages 122–132. Springer, 2016.

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

[71] Y. Xu, P. Zou, W. Qu, Z. Li, K. Li, and X. Cui. Sampling-based partitioning in mapreduce for skewed data. In *2012 seventh ChinaGrid annual conference*, pages 1–8. IEEE, 2012.

[72] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz. Wrangler: Predictable and faster jobs using fewer resources. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[73] T. Yeh and Y. Tu. Enhancing data availability through automatic replication in the hadoop cloud system. In *2018 9th International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*, pages 86–93. IEEE, 2018.

[74] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Osdi*, volume 8, page 7, 2008.

[75] W. Zeng, Y. Zhao, K. Ou, and W. Song. Research on cloud storage architecture and key technologies. In *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, pages 1044–1048. ACM, 2009.

[76] F. Zhang, J. Cao, S. U. Khan, K. Li, and K. Hwang. A task-level adaptive mapreduce framework for real-time streaming data in healthcare applications. *Future Generation Computer Systems*, 43:149–160, 2015.

[77] H. Zhang, H. Huang, and L. Wang. Mrapid: An efficient short job optimizer on hadoop. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 459–468. IEEE, 2017.

[78] Q. Zhang, S. Q. Zhang, A. Leon-Garcia, and R. Boutaba. Aurora: adaptive block replication in distributed file systems. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 442–451. IEEE, 2015.

[79] H. Zhou, Y. Li, H. Yang, J. Jia, and W. Li. Bigroots: An effective approach for root-cause analysis of stragglers in big data system. *IEEE Access*, 6:41966–41977, 2018.