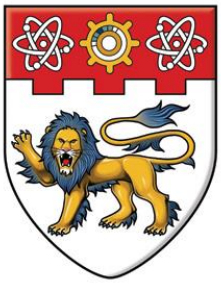

NANYANG TECHNOLOGICAL UNIVERSITY

Hadoop Job Scheduling with Dynamic Task Splitting

Xu YongLiang

School of Computer Engineering

2015

Hadoop Job Scheduling with Dynamic Task Splitting

Xu YongLiang

(G1002570K)

Supervisor

Professor Cai Wentong

School of Computer Engineering

This Thesis is submitted to the Nanyang Technological University in partial fulfillment of the requirement for the degree of Masters of Engineering

2015

Acknowledgements

First and foremost, I will like to give my heartfelt gratitude to my supervisor, Professor Cai Wentong, for his guidance and utmost patience throughout the course of my studies. His constant encouragement had been a great source of motivation for me especially in difficult times. His willingness to share his knowledge and spare his valuable time had been crucial in helping me meet the project's objectives.

I will like to thank Tang Shan Jiang from PDCC for spending his precious time with me to fine-tune the idea for my project. Special thanks to the lab technicians, Irene Goh and Chew Lerk Chern, for their assistance in the setup and maintenance of the cluster for my experiments in spite of their heavy workload.

I'm also grateful for the co-sponsorship of DSO National Laboratories and the support that my superiors had given me over the years.

I will also like to acknowledge the Fiona Low from SCE graduate office for her assistance for all administrative matters.

Lastly, I am grateful to my beloved family for the care, support and patience they have given to me over the years. I'm especially thankful to my wife whom spent many sleepless nights with me bouncing ideas with me and constantly pushing me to do my best.

Table of Contents

Abstract

Job scheduling affects the fairness and performance of shared Hadoop clusters. Fairness measures how fair the resources in the cluster are shared among different users in the Hadoop cluster. In Hadoop, schedulers will always attempt to maximize data locality. Data locality refers to the processing of data by tasks on nodes where the data is stored. Processing of data on data-local nodes improves performance, as there is no need to transfer data from one node to another.

However, fairness and data locality are often in conflict. During scheduling, it is not always possible that the available nodes contain the data that a user's job requires. In such cases, a scheduler may choose to schedule the tasks on these nodes regardless of data locality thus sacrificing performance. Alternatively, a scheduler may choose to give up the user's slot and wait for a data-local node thus sacrificing fairness. Achieving pure fairness may compromise the data locality of the tasks that will in turn negatively affects performances, and vice-versa. Delay scheduling is a technique that attempts to improve data locality by waiting for a data-local node to be available. It violates the fairness criteria.

The Dynamic Task Splitting Scheduler (DTSS) is proposed to mitigate the tradeoffs between fairness and data locality during job scheduling. DTSS does so by dynamically splitting a task and executing the split task immediately, on a non-data-local node, to improve the fairness. Analysis and experiments results show that it is possible to improve both fairness and the performance by adjusting the proportion of the task split. DTSS is shown to improve the makespan of different users in a cluster by 2% to 11% as compared to delay scheduling under conditions that is difficult to obtain data-local nodes on a cluster. Lastly, experiments show that DTSS is not a suitable scheduler under conditions where jobs are able to obtain data-local nodes easily.

List of Figures

List of Tables

List of Notations

ARIA	Automated Resource Inference Allocation
CDRM	Cost-Effective Dynamic Replication Management
DTSS	Dynamic Task Splitting Scheduler
DS	Delayed Scheduling
DARE	Distributed Adaptive Data Replication
DP	Dynamic Priority
ETL	Extraction, Transformation and Loading
HDFS	Hadoop Distributed File System
HOD	Hadoop On Demand
JVM	Java Virtual Machine
LSPS	Leveraging Size Pattern Scheduler
MIMP	Minimal Interference Maximal Productivity
MRCP	MapReduce Constraint Programming
PD	Performance Driven
SWIM	Statistical Workload Injection for MapReduce

Chapter 1. Introduction

1.1 Background

We live in the data age where the volume of data being generated increases every year. These data comes from a variety of sources such as social networks, server logs, documents scans, GPS trails, satellite imagery, financial markets etc. The influx of new data, which is also known as Big Data, provides a treasure trove of information in which organizations are able to exploit and benefit from. The success in the future of an organization will be dictated to a large extent on its ability to extract value from its collected data.

Big Data is characterized by three parameters, Volume, Velocity and Variety [1]. The attractiveness of Big Data is the volume of data that is available for analysis. However, the storage requirements for this huge amount of data exceed the current capabilities of traditional databases. This calls for a more scalable storage capabilities and efficient means to process the data. The velocity of the data, which is the rate of data flowing into an organization, also poses a challenge for organizations. Organizations will require the ability to quickly perform Extraction, Transformation and Loading (ETL) on the streams of data coming in. Additionally, organizations may also want to quickly work on the data as it is being collected in order to gain competitive advantage. It is normal for a Big Data system to utilize data from diverse sources. These data can be in text, imageries or any form of sensor information. Hence, there is a need for Big Data systems to support the storage of a variety of data. Analytical algorithms may also analyze data from different sources to gain insights not possible when some data sources are not available. One example of such application is Entity Resolution where the algorithm resolves a particular entity based on information from different source. For example, the algorithm can find out the different ways a person is addressed by linking information from different sources of data.

The advent of Big Data brings about many challenges to the existing data processing architecture. Hence, there arises a need for big data frameworks to complement the companies' existing infrastructure to support the analysis of the new data. There are many Big Data processing frameworks currently available in the market such as Dryad

[2], Apache Hadoop MapReduce [3], Apache Hadoop Yarn [4], and Apache Spark [5] to name a few.

1.2 Motivation

Hadoop is an open-source, reliable and scalable distributed processing framework meant for storage and processing of big data sets [3]. It provides companies with the capability to extract key information and discover new important insights. Considered as the platform of choice for Big Data processing and advanced analytics, Hadoop is adopted by many organizations such as Facebook and Yahoo [6].

Supported by the Hadoop Distributed File System (HDFS), Hadoop is able to recover from node failures and prevent data lost through data replication. Distributed parallel processing on Hadoop is performed through custom MapReduce data applications. These data analysis applications differ in terms of functionality, complexity and resource requirements, creating challenging situations for Hadoop in scheduling for better overall performance.

Job execution efficiency is an important aspect for Hadoop as it directly impacts the throughput of a Hadoop cluster. Hadoop relies on a job scheduler to make its scheduling decisions before executing tasks for any jobs. There are many schedulers available for Hadoop [7-18]. Based on the decision of the scheduler, a task belonging to the job will be selected for execution whenever a node is available. Hadoop considers the location of the input data for a task and tries its best to run on the node in which the data resides (data locality). When it is not possible to do so, Hadoop will make a copy of the data onto the selected node and schedule the task on the node. This operation creates overheads that may reduce the overall job execution efficiency.

Improving the data locality of tasks also improve cluster performance as it reduces the amount of overhead through reduction in the number of data transfers to other nodes for processing. Data replication enhances the data locality of tasks in addition to improving the fault tolerance of a Hadoop cluster. Through the creation of replicas, the probability that the input data is available on the node increases when a task using the data needs to be scheduled.

As organizations starts to embrace Big Data frameworks like Hadoop, there will be a growing need to share clusters between different users. As such, fairness of resource allocations will be an important aspect in Hadoop job scheduling. Scheduling resource fairly ensures that no one user of the cluster will hog the entire cluster resources leaving other users starved.

However, there is a conflict between data locality and fairness. For example, whenever a new slot opens up, the job that is selected for processing may not have data located at the node. In this situation, the scheduler may choose to proceed and schedule the job at this node thus compromising data locality for fairness. Conversely, the scheduler may choose to compromise fairness by scheduling another job's task whose data are located at the node.

1.3 Scope

The scope of the project is to study the conflict between data locality and fairness within a shared cluster through the scheduling of jobs. The study will be conducted on a Hadoop cluster. The Hadoop framework was used as it is adopted by many companies such as Facebook and Yahoo as mentioned above. Furthermore, one of the advantages of Hadoop is that it is able to run on commodity hardware. Hence, there is no need to purchase specialized machines in order to set up the cluster. The project will also study in detail, the architecture of a Hadoop cluster such as the workings of the Hadoop Distributed File System and the MapReduce paradigm. The state of the art in Hadoop job scheduling and data replication techniques will also be studied.

1.4 Overview of Report

The report will be organized as follows:

- Chapter 2 provides an overview of Hadoop. It will provide an introduction to the Hadoop Distributed Files System and the MapReduce paradigm
- Chapter 3 presents the state of the art in Hadoop job scheduling and data replication.
- Chapter 4 presents the Dynamic Task Splitting Scheduler (DTSS) that addresses the tradeoff between fairness and data-locality.
- Chapter 5 presents and discusses the experiment results obtained from DTSS.
- Chapter 6 concludes the report and suggests future works to be done.

Chapter 2. Hadoop Architecture

This chapter provides an overview of the Hadoop architecture and the inner workings of its individual components. This is to provide an understanding of the Hadoop's data management and its job scheduling workflow.

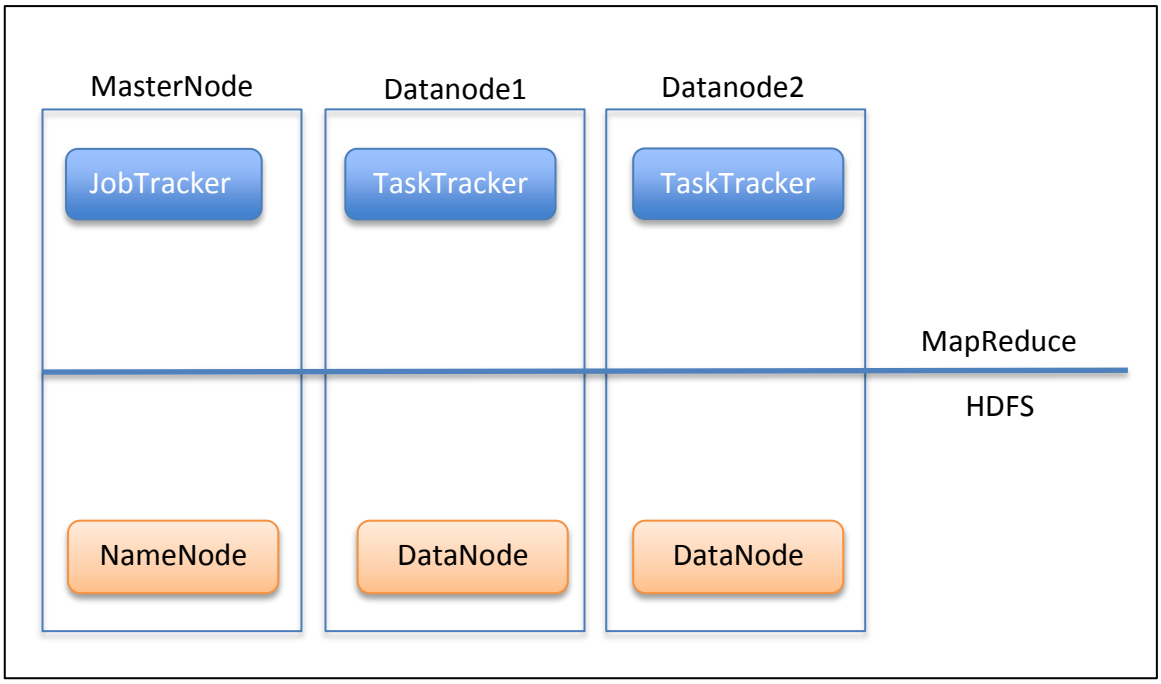

Figure 1 Hadoop Architecture

The Hadoop architecture consists of two key components; Hadoop Distributed File System (HDFS) and MapReduce. These components work collectively to deliver the reliability and scalability of a Hadoop cluster. Data is kept and managed by Hadoop Distributed File System while job executing and data processing are made possible using MapReduce. This section focuses on the inner workings of both HDFS and MapReduce.

2.1 Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) provides a fault tolerant solution for data storage in a Hadoop cluster. It is designed for the storage of large files with streaming data access patterns on clusters running on commodity hardware. This section will describe the Hadoop architecture followed by the concept of data blocks used within HDFS. The anatomy of read and writes operations for HDFS will also be described. The section will end with a description of the default replication scheme adopted in Hadoop.

2.1.1 Architecture

A HDFS cluster consists of two types of nodes that work in a master-slave configuration. The master node is known as the Namenode while slave nodes are known as Datanodes.

The key role of the namenode is to manage the namespace of the file system. It is responsible for maintaining the meta-data of each data block and to keep track of their location in the file system. The locations of each data blocks are kept in memory during the operation of the cluster and rebuilt whenever the cluster restarts. The namenode is crucial to the operation of the entire cluster since it is not possible to reconstruct a file using the individual blocks if the namenode fails.

The datanodes are where the data are stored in the cluster. It is responsible for the retrieval of the data blocks when requested by the namenode or client. It reports to the namenode through periodical heartbeats to update the list of data blocks that it is currently responsible for. The failure of a datanode will not cause the entire cluster to fail due to fault tolerance mechanisms built within the cluster.

2.1.2 Data Blocks

HDFS makes use the hard disks available in the nodes of the cluster. Storage in a physical disk is in terms of blocks, which is typically a few kilobytes. A block represents the minimum size in which data can be written and read from the disk.

Files in HDFS are broken into logical blocks that are distributed among the nodes in the cluster. However, the size of the data blocks in HDFS is in terms of megabytes to minimise the cost of seeks.

There are several benefits to having a block abstraction for HDFS. Firstly, it provides the capability to store a file that is larger than any disks available in the cluster. Secondly, it simplifies the management of the file system, as it is easier to compute the number of blocks that can be stored on a node and allows the meta-data of each block to be managed by the masternode. Lastly, data blocks can be insured against corruption and node failures easily through data replication.

2.1.3 Data Flow

a) Reading from HDFS

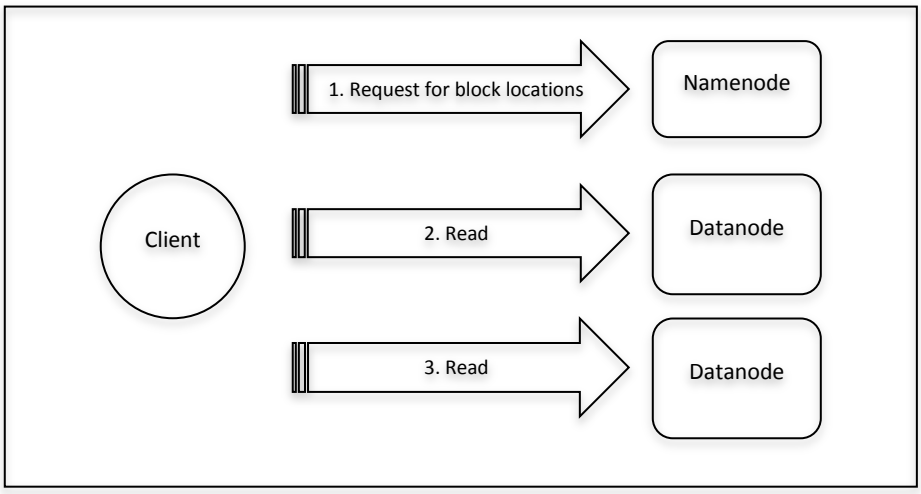

Figure 2 Reading from HDFS

Accesses to the data blocks of a file are performed independently of the namenode. Read requests are first communicated to the namenode to retrieve the locations of data blocks of the file requested. With the locations of the data blocks, clients are able to access the individual datanodes and retrieve the data blocks. The data blocks are retrieved individually from the nearest datanodes to the furthest based on the proximity of the nodes to the client machine. These are performed transparently to the client. Figure 2 shows the process in which data is read from HDFS.

b) Writing to HDFS

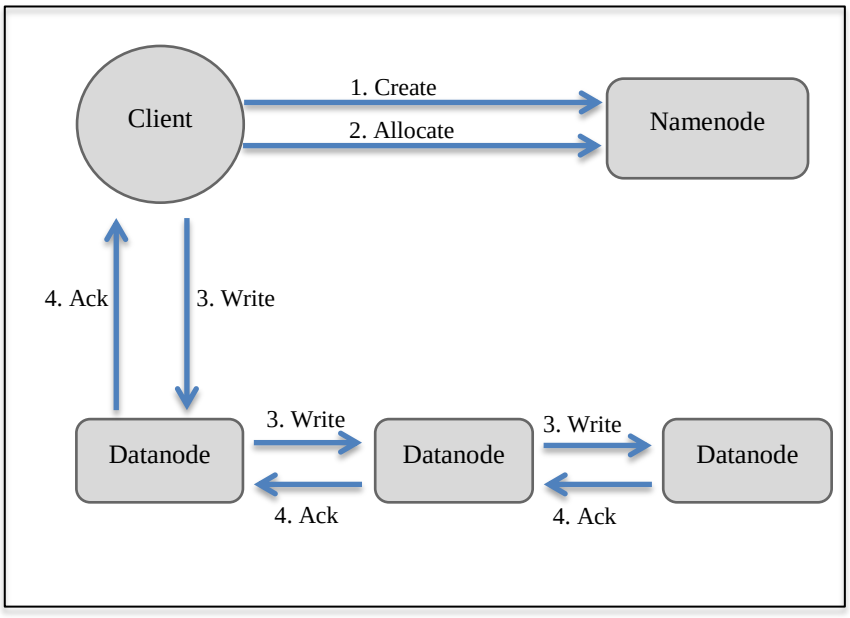

Figure 3 Writing to HDFS with 3 Replicas

Permissions are required from the Namenode before data can be written into the HDFS. Upon receiving the request, the Namenode ensures that the file does not exist and the

client has sufficient rights to create a new file. Next, the Namenode will create a record of the file in its namespace and informs the client to start its writing process.

Prior to writing the data, the client will request for the namenode to allocate new blocks. The namenode then selects a list of datanodes to store the data and informs the client of the selection. The number of datanodes selected by the Namenode depends on the number of replicas requested for the file. This list of datanodes is used to store the replicas of the file and forms a pipeline in which the client writes data. The client will stream the data block to the first datanode in the list, which stores the data and forwards it to the next datanode in the list and so on until the last datanode in the pipeline receives and writes the data. Acknowledgment is sent to the client after the last datanode successfully writes the data into its hard disks to ensure that the data is written to every datanode in the pipeline. The client repeats the writing process for the rest of the data blocks after receiving the acknowledgement until the entire file is written. Figure 3 shows the process in which data is written into HDFS, assuming the number of replicas is three.

HDFS adopts a write-once read-many access approach. Hence, a file once created, written and closed are immutable. This approach simplifies data coherency issues as clients can only read the data when it is closed.

2.1.4 Replica Placement

The placement of replicas needs to take into account the tradeoff between reliability and read/write bandwidth. Placing all the replicas on to a node will cause reliability problems while improving the writing speed since the write pipeline is within the node. Hadoop's default replica placement strategy is to place the first replica on the same node as the client if the client is running within the cluster. If the client is running off-cluster, a random node is chosen as the location for the first replica. The second replica will be placed on a random node in a different rack. The third replica will be placed in the same rank as the second replica but at a different node chosen randomly. Lastly, the rest of the replicas will be placed randomly within the cluster but the system will try to avoid placing too many replicas within the same rack.

2.2 MapReduce

MapReduce offers users the capability to perform large-scale parallalized data analysis on Hadoop clusters. It is a programming model that applications adopt to take advantage of

its many capabilities. Under this framework, applications do not need to worry about details such as parallelization, fault tolerance, load balancing and data distribution.

2.2.1 Architecture

The MapReduce framework consists of two key components; JobTracker and TaskTracker. There is only one JobTracker in the cluster. It is responsible for tracking all running MapReduce jobs executing in the cluster at any time. It performs scheduling operations and tracks each map and reduce tasks from all the jobs running across the nodes in the cluster. Tasks that fail will be reallocated to other nodes within the cluster.

There are multiple TaskTrackers in a Hadoop cluster. TaskTrackers are the workhorse in the MapReduce framework. They perform the map and reduce tasks that are assigned by the JobTracker. Every TaskTracker is configured with a fixed number of maps and reduce slots. These slots represent the numbers of concurrent map and reduce tasks that can be executed on the node. Each map or reduce tasks is spawned on a separate Java Virtual Machine instance so that any execution or process failure will not cause the TaskTracker to fail. TaskTrackers report their status and the tasks statuses to the JobTracker through periodic heartbeats. This allows the JobTracker to track the status of each TaskTracker and also provides JobTracker with information necessary to perform its scheduling operations.

TaskTrackers reside on the same node as the HDFS's Datanodes. An important feature in Hadoop is its ability to make use of its knowledge of the HDFS to schedule tasks to nodes in which the tasks' input data is located. This property is known as data locality. Since the input data is located in the same node as the tasks, reading of the data will be direct from the local disk, alleviating the strain on the network and reducing unnecessary network transfers. By bringing computation to the data, the overall performance will be improved since scheduling the task is faster than transferring the data to another node.

MapReduce is made up of two phases: Map and Reduce. Applications are required to implement its map and reduce functions to be used during the respective phases. Input data are fed into the map function as key-value pairs and processed to produce intermediate output also in key-value pairs. These intermediate output are then passed to the reduce function as inputs for processing and final output once the mapping functions are complete. The process which transfers the map outputs to the reduce task is called

shuffle. Inputs to the reduce tasks are sorted by key during the sorting phase before it is presented to the reduce tasks. Figure 4[19] shows the entire MapReduce process.

MapReduce input data is typically stored in the HDFS. Depending on the size of the input data, a MapReduce job may consist of multiple map tasks. Reduce tasks may be launched at different nodes in the cluster. Each of the map tasks is equivalent and can process any of the input files. Individual map task do not communicate with each other.

The next section describes the anatomy of a MapReduce job run.

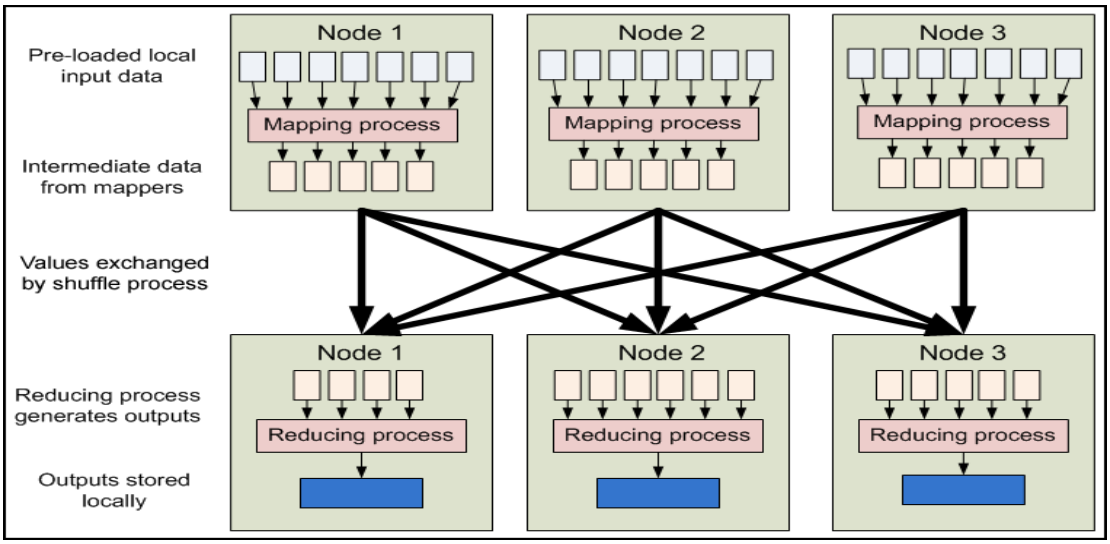

Figure 4 MapReduce Process

2.2.2 Anatomy of a MapReduce Job Run

The execution of a MapReduce job consists of many steps that are concealed to the user. It is imperative to be aware of the details of a MapReduce job execution in order to appreciate the complexity of operations behind the scenes. The MapReduce job execution consists of several phases:

- Job Submission
- Task Assignment
- Task Execution
- Job Completion

This section will describe the details during the various stages.

a) <u>Job Submission</u>

The job submission phase begins when a user submits a job. The first step of the submission process is to request for a new job ID from the JobTracker. After retrieving

the job ID, the job configuration is examined for errors such as invalid output folder. Input files are split into blocks equivalent to the block size defined for the file in the HDFS. The number of input splits is computed using the size of the input file and the block size. Each input split is assigned to a map task for processing. Hence, the number of map tasks for a job depends on the size and the block size of the data. It is important to balance the map tasks execution time and the amount of parallelism obtained. For example, having a high amount of parallelism may not be optimal if the execution time for a map task is very short as the overheads of setting up the high number of map tasks erodes the time benefit that is gained from the parallelism. Hence, the optimal approach is to ensure that a map task performs processing for a sufficiently long amount of time so as to reduce the overheads. One method to do this is to tune the block size of the file in the HDFS. After retrieving the input splits information, the job file and its relevant configurations will be copied over to the HDFS to prepare for task execution. These configurations are replicated across the HDFS to reduce the need to transfer them during tasks execution. The job submission process terminates after these data are successfully transferred to the HDFS. A job is queued when the JobTracker receives it.

b) Task Assignment

The task assignment phase handles the assignment of tasks to the respective TaskTrackers available in the cluster. Task assignments are under the purview of the JobTracker. TaskTrackers periodically sends heartbeats informing the JobTracker that it is alive and whether it is ready to accept new tasks.

The JobTracker needs to select a task to be assigned from the list of jobs that is currently executing or waiting for execution. Job selection depends on the scheduler used by the JobTracker. There are various scheduling algorithms available for Hadoop MapReduce clusters. More details on these scheduling algorithms will be discussed in Section 3.1 . As mentioned above, TaskTrackers are configured to have a fixed number of maps and reduce slots. The number of slots per node depends on the number of CPU cores and the amount of memory available on the node.

There is no requirement for data locality considerations for reduce tasks. Hence, they can be scheduled to run on any node as long as a reduce slot is available on the nodes. In contrast, when scheduling map tasks, it is important to ensure that the node being selected is as close to it input data split as possible for reasons mentioned above. Optimally, a task

will be scheduled on a data-local node. Task may be schedule on a rack-local node if it is not possible to be scheduled on a data-local node. It is only at the last resort that a task will be scheduled on a node on a different rack. The overall data localities of tasks in a Hadoop cluster can be improve through data replication. More details will be discussed in Section 3.2 .

c) Task Execution

After a TaskTracker is assigned a task, it will copy the required job files and its input data split (if not data-local) from the HDFS into its local file system. A new Java Virtual Machine (JVM) is launched for every task execution. This is to isolate the TaskTracker process to ensure that any crippling task failure will not cause the TaskTracker process to fail. Progress updates are sent periodically through its TaskTracker's heartbeats to the JobTracker.

d) Job Completion

A job is completed when all its tasks have successfully completed. The JobTracker will then notify the client if it is configured to do so. Intermediate data created during the sort and shuffle phases will be deleted.

2.3 Summary

An overview of the Hadoop architecture is presented in this chapter. Details on the different components of the Hadoop Distributed File System (HDFS) and the MapReduce framework are also discussed. Finally, the anatomy of a MapReduce job run is described. A review of the current state-of-the art in Hadoop scheduling and data replication will be presented in the next Chapter.

Chapter 3. Literature Review

3.1 Hadoop Job Scheduling

The scheduler plays a crucial role in managing the jobs submitted to the cluster for processing. It is the responsibility of the scheduler to select the job for processing each time a slot becomes available. The sequence in which the jobs are processed depends on the priority in which the scheduler placed on the following issues:

- Fairness among different users
- Improving response time
- Improving overall makespan of a set of jobs
- Meeting jobs deadline

Many schedulers have been proposed in the literature with each targeting at the different issues mentioned above. This section will first discuss about the FIFO Scheduler followed by the discussions on each of the issues mentioned above. The schedulers aimed at resolving the issues will be presented.

3.1.1 Hadoop Default Scheduler: FIFO Scheduler

The FIFO scheduler is a simple and straightforward scheduler that schedules jobs based on the first-in-first-out basis as its name suggests. Jobs are serviced based on their arrival time regardless of the job lengths. The FIFO scheduler supports up to five priority levels which are used for prioritizing the jobs. Jobs with higher priority level are always serviced first regardless of the amount of time the other jobs have been waiting. No pre-emption of jobs is supported.

While the FIFO scheduler is adequate in fulfilling the requests for multiple jobs efficiently, it is not optimized to serve the needs of a multi-tenant Hadoop environment. In a multi-tenant environment, multiple users (tenants) share the same cluster resources for execution of their data processing jobs. Scheduling the jobs in sequence purely based on time of arrival and priority levels do not provide users any guarantees in terms of fairness, response time, deadlines adherence or isolation from other users. A large job from a user with high priority will hog the cluster resources for a significant amount of time resulting in starvation of other jobs. Likewise, huge number of job submissions from users with higher priority will also result in starvation for users of lower priorities.

The overall data locality of the cluster when using FIFO scheduler may also suffer as it only looks into scheduling tasks belonging to the current job whenever a slot becomes available. For example, the node holding a free slot may contain the data belonging to the other jobs in the queue but not the input data for the current job. In spite of this, task from the current job will still be scheduled on the non-data local node thus reducing the overall data locality of the cluster.

The limitations with the FIFO scheduler make it unsuitable in situations where there is a need to maintain some form of service standards for jobs submitted by different users. The handling of data localities of tasks is also suboptimal and can be improved by considering other jobs in the queue during scheduling.

3.1.2 User Isolation

Isolation of users is useful when there are requirements to ensure a minimum share of cluster resources to different groups of users. For example, a Hadoop cluster may be shared across different departments within a company with each department having different computing needs. Hence, each department may be assigned to different amount of cluster resources.

3.1.2.1 Capacity Scheduler

The Capacity Scheduler provides a way to share a Hadoop cluster with multiple users or groups with capacity guarantees [8]. To support multiple users, the scheduler splits the cluster into multiple queues. Each queue has an allocated capacity that will be made available to all the jobs sent to the queue. The capacity allocated to each queue is pre-defined by the cluster administrator and can be adjusted by the administrator at runtime. Multiple users may submit jobs to a queue. To prevent hogging of queue resources by a user, each user can only utilize up to a pre-defined amount of resources.

Job scheduling within each queue is performed independently of each other. Within each queue, jobs with higher priority are given access to the queue resources. The scheduler does not support task preemption. However, preference will be given to tasks from higher priority jobs if available in the next scheduling epoch. To improve the utilization of the cluster resources, queues are provided access to the resources beyond their capacity through the usage of the idle resources from other queues. The resources are returned after the existing tasks execution when the demand of other queues picks up.

Capacity scheduler provides the functionality to schedule tasks based on the memory requirements of a job, as it is not limited to the one slot per task paradigm. If required, a task can request for more than a single map/reduce slot (more resources) to process its map/reduce tasks. This capability provides the flexibility to run memory intensive jobs that may not be possible previously.

3.1.2.2 Hadoop On Demand

Hadoop On Demand (HOD) [13] provides an alternative solution to isolate users from one another by provisioning the cluster into multiple private ones for different users. The amount of resources allocated to each of the private cluster is at the discretion of the cluster administrator. The private clusters share the same file system and have access to the same data set. Similar to the Capacity scheduler, jobs scheduling is performed independently within each private cluster. However, HOD is not as flexible as the Capacity scheduler in terms of resource management. In HOD, each private cluster has exclusive access to their assigned resource. Un-utilized resources are not accessible to other private clusters unlike the Capacity scheduler. This creates situations in which other users may be starved of resources even when resources are readily available in the other private clusters, and thus reducing the overall cluster utilization.

3.1.3 Fairness

The Capacity Scheduler and HOD does not consider issues such as fairness of resource allocation among different users who are sharing the cluster. Resource allocation in the Capacity Scheduler and HOD are fixed based on the users or groups as mentioned above. These strategies are suitable only if the workloads of the users are well defined. In cases where the workloads of the users are unknown and subjected to huge deviation, it is more appropriate to assign each user a fair share of the cluster. This also allows the scheduler to dynamically reallocate unused resources to other users. This section discusses schedulers that deal with fairness in scheduling of Hadoop clusters.

3.1.3.1 Fair Scheduler

The Fair Scheduler aims to provide every user a fair share of the cluster over time [15]. Like the Capacity Scheduler and HOD, the Fair Scheduler also isolates users from one another, providing an illusion of owning a private cluster. Under the Fair Scheduler, the cluster is divided into pools, one for each user of the cluster. Users of the cluster submit their jobs to their respective pools for processing. Similar to the Capacity Scheduler, the scheduling of the job is performed independently within each pool. Each pool is

guaranteed a minimum share of resources as long as there is demand within the pool. The minimum share for each pool, which can be zero, is pre-defined by the cluster administrator. The sum of the minimum shares for the pools must not exceed the capacity of the cluster. To maximize the overall resource utilization, other pools can utilize resources from pools that are not using their full minimum share.

Slot allocation in the Fair Scheduler happens dynamically when there are changes in the demand within each pool. The initial slot assignment process ensures that the minimum share of each pool is fulfilled and unused resources from underutilized pools are shared. The process begins by first assigning slots to pools whose demand is less than its minimum share. Next, slots are allocated to the remaining pools up to their minimum share ensuring that the minimum share criteria for the pools are met. Finally, the slots that are not utilized are distributed equally to pools that have unfulfilled demands.

Slot reallocations are performed when job submissions to the pools increase to ensure that the minimum share criterion for each pool is met. The minimum share is the minimum amount of resource required for the pool. During slots reallocation, the Fair Scheduler terminates tasks from jobs that are running using the slots 'borrowed' from other pools. The Fair Scheduler utilizes task killing timeouts T_{min} and T_{fair}. The timeout T_{min} represents the maximum time to wait for a pool to receive its minimum share. If a pool does not receive its minimum share by this timeout, tasks will be killed to fulfill the pool's minimum share. T_{fair} represents the maximum time to wait for a pool to receive its fair share. Likewise, if a pool does not receive its fair share by T_{fair}, tasks will be killed to allow the pool to receive its fair share of the cluster. These thresholds allow the pools and jobs to wait for slots to be available before initiating task termination, as the terminated tasks have to be reassigned and processed from the beginning if killed. This process aims to reduce computation wastage that accompanies the termination of an incomplete task. To further minimize wastages, the scheduler kills tasks that are most recently launched as opposed to tasks that have been executing for some time.

3.1.3.2 Dynamic Priority Scheduler

Static allocation of resources (Capacity Scheduler and HOD) and the guarantee of minimum share (Fair Scheduler) do not accurately represent a user's willingness to acquire computation resources to service their needs. These approaches require the administrator of the cluster to pre-define the amount of resources allocated to each user.

Once fixed, the users are not able to optimize the usage of their granted allocation across jobs of different importance.

The Dynamic Priority Scheduler (DP) dynamically allocates resources to users based on their willingness to purchase higher priority during scheduling [10]. Under the DP Scheduler, users are granted a budget that they use to purchase Map and Reduce slots per unit time. The time unit is known as the allocation interval. Resources are dynamically distributed to users at each allocation interval based on their bids for the available resources. This model is biased towards users with small jobs as they are able to outbid users with larger jobs. Given the same budget, users with smaller jobs will be able to complete more jobs due to the smaller resource requirements per job. Hence, the users with larger jobs have scale down their resource consumption in order not to overspend. By scaling down their consumption, larger jobs will take longer time to complete as fewer tasks can be run concurrently.

The aim of this scheduler is to allow users to dynamically manage their resource consumption, not to isolate the users from one another like the Capacity Scheduler, HOD and Fair Scheduler. Like the Fair Scheduler, spare resources are also distributed to other jobs in order to improve the overall throughput of the system. Pre-emption of tasks running on the spare resources occurs when resource allocation changes.

With the DP Scheduler, there will be more variation in the capacity allocated to a user's job due to the bidding of slots. Hence, it is difficult to maintain a constant capacity for a user's job since the capacity obtained by a job depends on the bids submitted by all users of the cluster at each allocation interval. The variation in the capacity allocated to a user's job increases the difficulty to meet a job's deadline.

3.1.3.3 COSHH Scheduler

The COSSH scheduler [20] considers the scheduling of jobs in heterogeneous Hadoop clusters. The scheduler consists of three processes: a task scheduling process, a queuing process and a routing process. The task scheduling process is initiated when a new job is submitted to the cluster. During the task scheduling process, the scheduler uses a task duration predictor to estimate the mean execution time of the job. The estimated execution time is then used at the queuing process. The queuing process consists of many job queues classified by the jobs' priority, mean execution time and mean arrival rate.

Classification is achieved through k-means clustering [21]. Unlike the Fair Scheduler, where jobs from users will only be assigned to their own pool, the queues in COSSH are shared across all users.

The routing process is initiated when a free slot is available. It requests for a set of suggested classes from the queuing process and select a task from the classes. The queuing process suggests a set of classes through an optimization process that tries to minimize the overall execution time. The optimization process considers the properties of each class and the features of the available resources (slots execution rate, memory available etc). Fairness is achieved by selecting jobs from users that have yet to achieve their minimum share. Jobs from users that require the most slots to achieve their minimum share will be give priority.

3.1.4 Response Time and Makespan Reduction

The aim of Capacity Scheduler and HOD's is to provide user isolation in a multi-tenant Hadoop cluster environment while Fair Scheduler attempts to provide a fair share of the cluster resources to users. However in these schemes, the resource allocations for different users do not consider the diversity of the users' workloads. Although the DP Scheduler provides the ability to dynamically distribute resources, it is still dependent on the users to manage their own resource usage. These schedulers do not consider the response time of the cluster or the makespan of the jobs in cluster as a criterion during scheduling.

The response time of the cluster represents the time it takes for a job to be submitted to the time it completes and the information is returned to the user. By reducing the response time for each job, the scheduler will be able to make the cluster more responsive for individual users thus improving the user experience. The makespan of a set of jobs represents the overall time it takes to complete the processing of these jobs. Changing the order of their processing can vary the makespan of a set of jobs. Improving the makespan will result in the improvement of the throughput of the cluster. For a set of jobs, if the average response time for the jobs in the set is short, the makespan will also be short. This section will look at the LsPS Scheduler that is aimed at improving the response time of the cluster and the BalancedPool Scheduler that looks at makespan reduction.

3.1.4.1 Leveraging Size Pattern Scheduler

The Leveraging Size Pattern Scheduler (LsPS) [22] looks at improving the overall job response time by dynamically adjusting resource allocation and scheduling policy among users based on historical knowledge of workload patterns. It tracks important statistical information of each job to build up historical data on the workload patterns of each user. This information is then used for slot allocation and selection of the scheduling policy to be used. The scheduler is made up of two-tiers of scheduling schemes (similar to the Fair Scheduler) based on the present job size patterns of each user.

In the first tier, the scheduler allocates slots to users according to their workload. Slots allocation for each user is computed using the statistical information based on the historical data collected. The average job size of a user is shown in (1):

$$\bar{S}_i = \frac{1}{|J_i|} \times \sum_{j=1}^{|J_i|} n_{i,j} \times \bar{t}_{i,j} \quad (1)$$

where $J_i$represents the set of jobs from user i that are current running or waiting for service, $n_{i,j}$ represents the tasks number of the $job_{i,j}$ and $\bar{t}_{i,j}$ represents the average task execution time of the $job_{i,j}$.

After computing the average job size of a user, the LsPS scheduler updates the slot allocation adaptively as shown in (2):

$$SU_i = SU_i^* \times U \times \left(\alpha \times \frac{\frac{1}{\bar{S}_i}}{\sum_{i=1}^{U} \frac{1}{\bar{S}_i}} \right) + (1 - \alpha) \times \frac{1}{U} \quad (2)$$

where SU_i^* represents the number of slots for user i if slots are shared equally among all users, U represents the number of users that are active in the system and α is a tuning factor used for controlling how much the scheduler biases users with small jobs. α ranges from 0 to 1. When α is close to 0, the slot allocation will be fairer. When α is close to 1, the scheduler will assign more slots to users with smaller jobs.

It is guaranteed that each user in the cluster will be assigned at least a slot (3). This prevents any users from being starved for processing slots. Equation (4) suggests that all slots in the cluster will be fully assigned to all active users.

$$\forall i, SU_i > 0 \quad (3)$$

$$\sum_{i=1}^{U} SU_i = \sum_{i=1}^{U} SU_i^* \quad (4)$$

In the second tier, the scheduler selects either the FIFO or Fair scheduling scheme to schedule jobs belonging to each user. Based on the experiments performed by Yi Yao et. al., the FIFO scheme is superior to the Fair scheme when the jobs submitted to the cluster have similar size [22]. Therefore, LsPS makes use of the historical workload data from each user to decide which schemes to use for the scheduling of jobs for the user.

In summary, LsPS dynamically adjusts the number of slots according to the statistical information regarding each user's workloads (first tier) while the job scheduling scheme for each user will be dynamically determined by the user's average job size (second tier).

3.1.4.2 BalancedPool Scheduler

The sequences in which jobs are processed have significant impact on the overall job completion time (makespan) and cluster utilization. The goal of the BalancedPool Scheduler [18] is to optimize the order in which a set of MapReduce jobs is processed to reduce their overall makespan. The scheduler partitions the jobs into two pools, one for small jobs and the other for big jobs, such that the makespans of the pools are balanced and the overall completion time in both pools are minimized. The Johnson's Algorithm [23] is applied to create a schedule for the jobs within each pool. The schedules are then tested using a MapReduce Simulator, SimMR [24] to get the approximation of the makespans. The scheduler searches for the minimal makespan by shifting jobs between the pools.

By improving the response time for each user, the LsPS Scheduler reduces the time it takes for a job to complete such that the results can be returned to the user at a faster rate. However, reducing the response time might not necessarily results in better cluster utilization. Therefore, the BalancedPool Scheduler adopts a more direct approach by reducing makespan of a set of jobs to improve the overall cluster utilization. Through reduction in makespan, jobs will be completed in shorter time thus allowing the cluster to process more jobs.

3.1.5 Deadlines

With limited cluster resources, it is beneficial for the users to specify the deadline requirements for their jobs so as to improve the resource allocations. Specifying timing requirements for each job that is submitted to the cluster allows for jobs to be scheduled to meet these requirements. This results in a more efficient allocation of resources. Specifying jobs priority is not as effective since many jobs from different users may hold the same priority. This section discusses schedulers that are designed to meet the deadline objectives of individual jobs.

3.1.5.1 Performance-Driven Scheduler

The Performance-Driven (PD) Scheduler [17] relies on the estimations of a job's completion time given a specific resource allocation to meet its performance target. The scheduler based on the estimations adjusts the resource allocation dynamically.

The scheduler computes the mean completed task length for a running job m as shown in (5):

$$\mu_m = \frac{\sum_{i \in C_m} \alpha_i^m}{|C_m|} \quad (5)$$

where $C_m$refers to the number of completed map tasks and $\alpha_i^m$is the time it takes for a task $t_i^m$to complete. This equation applies for both map and reduces tasks.

Slot assignments for each job are dependent on its deadline and the estimated amount of work still pending. The number of slots to be assigned is estimated as shown in (6):

$$s_{req}^m = \frac{\left(\frac{\sum_{i \in R_m} \delta_i^m}{\mu_m} + |U_m|\right) * \mu_m}{T_{goal}^m - T_{curr}} - |R_m| \quad (6)$$

where δ_i^m is the amount of time remaining for a task, T_{goal}^m is the deadline for the job, T_{curr} is the current time and R_m is the number of tasks from job m which are currently running. U_m is the number of tasks not yet started. The number of slots for both map and reduce tasks is computed in the same manner.

Based on (6), the number of slots assigned for a job increases when any of the following conditions is met:

a) the amount of time remaining for a task (δ_i^m) is high
b) number of tasks not yet started (U_m) is high
c) when the deadline is nearing

Unlike LsPS that keeps historical workload data for different users, the PD Scheduler only tracks a job's tasks during its execution lifetime. As such, the scheduler will need to give priority to newly submitted jobs in order to collect the required workload information for scheduling purposes. As the PD Scheduler is a best-effort scheduler, it is possible for jobs to miss their deadlines. Hence, priority will be given to these jobs whose deadline has been missed.

3.1.5.2 Automatic Resource Inference and Allocation Scheduler

Like the PD Scheduler, the Automatic Resource inference and Allocation (ARIA) [7] Scheduler's goal is to dynamically adjust the resource allocation for the jobs to meet the timing requirements (soft deadline). The ARIA Scheduler tracks a job's execution information to build up a job profile. However unlike the PD Scheduler, this information is persisted and used for future scheduling. The premise for persisting job information is that production jobs are often executed periodically on new datasets. As such, it is more efficient to constantly maintain the job information instead of building the information from scratch like the PD Scheduler.

The ARIA Scheduler estimates the completion time of the job as shown in (7):

$$T_J^{low} = M_{Avg} \times \frac{N_M^J}{S_M^J} + B_J^{low} \times \frac{N_R^J}{S_R^J} + C_J^{Low} \quad (7)$$

where $M_{Avg}$is the average duration of a map task, $N_M^J$and $N_R^J$are the number of map tasks and reduce tasks for job j respectively. S_M^J and S_R^J are the number of map and reduce task slots assigned to job j respectively. $B_J^{low}$is the average duration of a reduce tasks plus the average time for shuffling data to the reduce tasks while $C_J^{Low}$is the time difference between the first shuffle phase duration and the average shuffle phase duration. The first shuffle phase always takes a longer time compared to the rest of the shuffle phases as it overlaps with the entire map phase. Based on (7), the completion time of a job increases

when the number of map/reduce slots assigned to the job is lower than what the job requires.

The number of slots for a job is found by equating the above function to the deadline (T) then using the Lagrange's multipliers to compute the minimum value for $S_M^J$and S_R^J. The scheduler executes the Earliest Deadline First algorithm to decide the job ordering during slot allocation.

3.1.5.3 MapReduce Constraint Programming Based Hadoop Scheduler

Unlike the PD scheduler and ARIA, the MapReduce Constraint Programming scheduler (MRCP) utilizes constraint programming to optimize the scheduling of MapReduce jobs [25]. It formulates the MapReduce scheduling into an optimization problem with an objective function and a few constraints.

The objective function of the scheduling problem focuses on the minimization of the number of late jobs. A few constraints are defined:

a) one task can only be assigned to one resource to prevent over allocation of resource
b) each assigned map start time is after its job's start time to ensure that the solution does not start a map task before a job is started
c) reduce tasks can only start after all map tasks are complete
d) completion time of a job is set to the completion time of the last reduce task
e) assigned map tasks and reduce tasks do not exceed the capacity of the cluster

MRCP requires estimations of the tasks completion times to perform the optimization. The estimation is obtained through analysis of historical logs from previous runs and is submitted to the scheduler by the user as part of the job submission. At each scheduling interval, MRCP will evoke the IBM CPLEX solver to solve the constraint optimization problem. Assignment of tasks is performed based on the recommendation from CPLEX.

3.1.5.4 Minimal Interference Maximal Productivity Scheduler

A Hadoop cluster may consist of a mix of dedicated and shared nodes. Shared nodes are created through virtualization of the machines while dedicated nodes are machines dedicated to running Hadoop processes. Sharing a node with other users may inversely

affect the performance of the Hadoop processes on the node. The Minimal Interference Maximal Productivity (MIMP) scheduler [26] aims to minimize the effect of interference on the Hadoop processes on the shared nodes and to ensure that deadlines are met for multiple jobs.

MIMP consists of two schedulers: MI Scheduler and MP Scheduler. The MI scheduler is responsible for the scheduling of virtual machines. It prevents the low priority virtual machines from interfering with the higher priority virtual machines. The MP scheduler is responsible for the deadline-aware scheduling. The scheduler uses task completion time models to predict task completion time on nodes with different resources. The task completion time model is trained by executing jobs on nodes with varying CPU capabilities. Using the predicted task completion time, the scheduler will then estimate if a job will be able to meet its deadline. Jobs that are likely to miss their deadlines are given priority during scheduling.

3.2 Hadoop Data Replication

Replication schemes consider the number of replicas to create, where to place the replicas and when to create the replicas. The decisions made with respect to each item impact the overall cluster throughput. For example, having a replication factor equal to the number of nodes in the cluster will achieve 100% data locality but significantly reducing the amount of storage available in the cluster, thus making it infeasible practically. As the number of replica increases, the writing speed of files into HDFS will be reduced due to the pipeline approach that Hadoop adopts for replica creation. Replica placements affect the overall (data and task) load balancing of the cluster. Placing replicas at nodes that are busy or overloaded will not improve the data locality of jobs due to slot contention issues. In the fixed replication scheme, replicas are created whenever a file is copied into the HDFS. In contrast, dynamic replication schemes will need to take into consideration whether to create the replicas passively or actively. In passive data replication, replication occurs only when there is a request to increase the number of replications of a file. On the other hand, active data replication schemes adjust the number of replications actively at a scheduled time. This section discusses the dynamic replication schemes available in the literature for Hadoop clusters.

3.2.1 Scarlett

Scarlett's [27] data replication policy is dependent on the popularity of the data. It works by computing a replication factor for each file that is proportional to it popularity. To do so, Scarlett counts the maximum number of concurrent accesses (c_f) to a file over a learning period (typically 24 hours). Scarlett replicates the files proportional to c_f. The replication factor for each file is computed as shown in (8):

$$max(c_f + \delta, 3) \quad (8)$$

where δ is used to cushion against under-estimates and three replicas is the lower bound to ensure that the performance is at least on par with the default HDFS setting. Scarlett performs adjustments to the replica numbers at a pre-defined interval.

Adjustments to the replication number for each file are dependent on the replication budget that is available on the system. The purpose of the replication budget is to limit the amount of storage space used by the replicas. Scarlett uses a round-robin approach to ensure fairness over the allocation of the budget. It works by increasing the replication count for each file by one iteratively until the budget runs out.

Scarlett attempts to place replicas for each file on as many datanodes as possible to improve the computational load balancing across the cluster. Adjustments begin by first de-replicating files that are over replicated and updating the budget in order to get an updated view of the cluster. Files are considered over replicated when their popularity do not justify their existing replication levels. An expected load for each datanode within the cluster is then computed and used for assigning replicas to nodes. Nodes with lesser load will be prioritized during the assignment.

3.2.2 Distributed Adaptive Data Replication

Like Scarlett, the Distributed Adaptive Data Replication (DARE) [28] algorithm also works based on the concept that popular files should be replicated more freely in order to improve the overall data locality of the cluster. Popular files are frequently accessed by jobs in the cluster and affect the overall data locality of the cluster, as the nodes storing the data will be constantly overloaded computationally.

DARE addresses the data replication issue on two fronts: replica allocation and replica placement. Replica allocation refers to the process to decide what data to replicate. Replica placement refers to the process to decide on which node to place replicas.

DARE uses a probabilistic approach for replica allocation which prevents a high rate of replica creation and eviction known as thrushing. Replicas are dynamically created with a probability when tasks are scheduled to run on non-data local nodes. DARE rides on Hadoop's data retrievals whenever a data needs to be copied to a non-data local node for processing. However, instead of deleting the data after a task is completed, DARE retains the data as a replica for future use. This approach permits the creation of replicas without consuming extra network and computation resources.

Similar to Scarlett, DARE also uses a replication budget to limit the storage that is consumed by the dynamically created replicas. Replica evictions happen when the allocated budget is exceeded and there is a need to make space for new replicas. The selection of replicas to be evicted is based on their access counts so that less popular data will be evicted first.

Under Scarlett, the replication adjustments happen at a pre-defined interval. In contrast, replication adjustments happen dynamically under DARE during scheduling. This allows DARE to react more quickly to changes in file popularity.

3.2.3 Cost-Effective Dynamic Replication Management

The availability of the data in a Hadoop cluster depends on the number of replicas and the popularity of the data. The availability of data increases with the number of replicas. The Cost-Effective Dynamic Replication Management (CDRM) scheme uses a model to capture the relationship between availability of data and the replica number [29]. Through adjustments of the replica number and their location, CDRM can dynamically distribute workloads across data nodes to improve performances.

The namenode in HDFS computes the minimum number of replicas required in order to satisfy an expected availability given an average data node failure rate. Replica placements depend on a blocking probability that is used as a criterion so as to improve load balancing and parallelism. The blocking probability is used to decide where to place the replicas. The blocking probability of a datanode i is computed as shown in (9):

$$BP_i = \frac{(\lambda_i \tau_i)^{c_i}}{c_i} \left[\sum_{k=0}^{c_i} \frac{(\lambda_i \tau_i)^{c_i}}{k!} \right]^{-1} \quad (9)$$

where λ_i represents request arrival rate for the data node, τ_i represents the average duration the datanode serves a request and c_i represents the number of slots available in the datanode i. The values of λ_i and τ_i are computed using the historical access information for each data block and used to compute the blocking probability periodically. Data replicas will be placed on datanodes with the lowest blocking probability.

3.3 Fairness versus Data Locality

Fairness and data locality are two opposing factors in Hadoop job scheduling. In a Hadoop cluster shared by multiple users, it may not be always possible to schedule tasks to its data-local nodes. Following a strict data locality policy may cause users to lose their chance to run their tasks, as they do not have data on the available machine. This policy sacrifices fairness in favor of data locality. Conversely, using a strict fairness policy will negatively affect data locality, as the scheduler will schedule users' tasks regardless its data locality.

3.3.1 LSAP Scheduler and LSAP-Fair Scheduler

The LSAP scheduler [30] considers all the available slots in the cluster during scheduling. It uses a cost matrix to capture the cost of data staging for scheduling jobs to yield optimal data locality. Each entry in the matrix represents the assignment cost of all possible task assignments to each available slot in the cluster. The scheduler attempts to find task assignments that yield the lowest overall cost using the cost matrix by modeling the task assignments into a Linear Sum Assignment Problem (LSAP). This scheduler is shown to out-perform delay scheduling when there are more idle slots and tasks. However, when the cluster is operating at maximum capacity, the performance advantage of the scheduler will be attenuated.

The LSAP-Fair scheduler [31] extends the LSAP scheduler by integrating fairness. The scheduler does so by dividing the assignment cost into two parts consisting of the fairness cost and the data locality cost. This allows administrators to tune the scheduler such that it favors fairness or data locality or a balanced policy.

3.3.2 Delay Scheduler

Delayed scheduling [9] is a technique used to improve data locality in a Hadoop cluster. Under delay scheduling, when the job to be scheduled does not have data-local task on an available node, it will be skipped and other jobs will be given the chance to be scheduled. However, the job will be scheduled regardless of data-locality to avoid starvation if it has been skipped for too long

The key insight of delay scheduling is that the chance of a job getting a data-local node is lower at the first try but improves after waiting for a short amount of time. The reason for this is that typical Hadoop tasks are short relative to jobs and multiple copies of the same data block are stored in the cluster. The higher the numbers of replicas of the same data block relative to the number of nodes, the higher the chance to obtain a data-local node during scheduling. For example, for a cluster with N nodes, 100% data locality will be guaranteed if the number of replicas for all data is set to N.

Delay scheduling negatively impacts fairness in favor of performance. Waiting for a data-local node compromises fairness as the user foregoes its chance to process its tasks. However, a strict implementation of fair sharing will also compromise performance, as tasks will be scheduled regardless of data locality.

3.4 Summary

Scheduling of jobs plays a crucial role in a Hadoop cluster. It is responsible for the selection of jobs to process whenever a task slot becomes available. There are several challenges of job scheduling in Hadoop. Various job-scheduling techniques each solving specific challenges in Hadoop scheduling were presented in this chapter (Table 1).

Challenges	**Schedulers**
User Isolation	Capacity Scheduler [8] HOD Scheduler [13]
Fairness	Fair Scheduler [9] DP Scheduler [10] COSSH scheduler [20]
Makespan Reduction	LsPS [22] Balanced Pool Scheduler [18]
Deadlines	PD Scheduler [17] ARIA Scheduler [7] MRCP Scheduler [25] MIMP Scheduler [26]

Table 1 Schedulers reviewed in this chapter.

Hadoop uses a simple FIFO scheduler for its scheduling purposes. While the FIFO scheduler is adequate for fulfilling the needs of multiple jobs it is not meant for a multi-tenant environment. The data locality of the cluster when using FIFO scheduler may also suffer as tasks from the current job will be scheduled on the first available node even if none of the tasks have data on the node.

Unlike the FIFO scheduler, the Capacity scheduler and HOD scheduler are tailored specifically for multi-tenants purposes. The Capacity scheduler shares the cluster resources among multiple users by splitting the resources across multiple queues. The jobs in each queue are scheduled independently. Unlike the FIFO scheduler, the Capacity scheduler provides the flexibility to assign more than a single slot to tasks that requires more resources. Compared to the Capacity scheduler, the HOD scheduler is more rigid in resource allocation. In the HOD scheduler, resources are provisioned into multiple smaller private clusters. These resources once allocated are assigned exclusively to the users of the private clusters. Hence, un-utilized resources within each private cluster are inaccessible to other private clusters. Table 2 compares the two schedulers meant for user isolations reviewed.

Scheduler	**Priority**	**Pre-emptive**	**Allocation**
Capacity Scheduler	Yes	Yes	Dynamic
HOD Scheduler	No	No	Static

Table 2 User Isolation Schedulers

In a multi-tenant Hadoop cluster, it is also important to ensure that resources are distributed fairly among the different users. The Fair scheduler and DP scheduler are designed specifically for this purpose. The Fair scheduler aims assign resources such that all users get the same amount of resources overtime. Similar to the Capacity and HOD schedulers, resources are divided into pools that guarantee minimum share resources for users. Un-utilized resources are made available to other users similar to the Capacity scheduler. The Fair scheduler uses pre-emption of tasks to ensure that users get their fair share of the cluster at reasonable speeds. This is different from the Capacity scheduler, which does not support task pre-emption. Instead of sharing cluster resources equally among different users, the DP scheduler uses a bidding mechanism to allocate resources to users whom are allocated limited budget which acts as currency. Although the DP

scheduler allows the users to bid for resources depending on their needs, it also creates a variation in the resource allocations for jobs. Similar to the Fair scheduler, COSSH also attempts to meet the minimum share requirements for each user. Table 3 compares the fairness schedulers reviewed.

Scheduler	**Pre-emptive**	**Resource Allocation**	**Minimum Share**
Fair Scheduler	Yes	Dynamic	Yes
DP Scheduler	No	Dynamic	No
COSSH Scheduler	No	Dynamic	Yes

Table 3 Fairness Schedulers

The aims of the schedulers mentioned above are to achieve users isolation (Capacity and HOD) or to achieve fairness in resource allocation (Fair, DP and COSSH). However, these schedulers do not take into consideration the response time of the cluster or the makespan of jobs in cluster as a criterion for scheduling. LsPS looks at improving overall job response time through dynamic adjustments of resource allocation based on historical workload patterns. On the other hand, the BalancedPool scheduler looks at improving the makespan of a set of jobs. Table 4 compares the makespan schedulers reviewed.

Scheduler	**Resource Allocation**	**Scheduling Technique**
LsPS	Dynamic	Statistical
BalancedPool	Pre-processed	Simulation

Table 4 Makespan schedulers

The PD scheduler and ARIA scheduler are specially designed to perform scheduling to meet the deadline of jobs submitted to the cluster. The PD scheduler relies on estimations of a job completion time to dynamically adjust resource allocations to meet the job's deadline. It does not keep historical records of the jobs unlike the ARIA scheduler. The ARIA scheduler works based on the premise that production jobs in a cluster are often executed periodically on new datasets. Hence, in cases where such premise do not hold, the PD scheduler is perhaps more suitable. MRCP models the task scheduling as a constraints programming problem and treats task scheduling as an optimization problem. The MIMP scheduler aims to minimize the effect of interference on the Hadoop processes on the shared nodes and to ensure that deadlines are met for multiple jobs. Both the MRCP and MIMP do not track historical task run times. Table 5 compares the deadline schedulers reviewed.

This chapter also covered the current state of the art in dynamic replication schemes used in Hadoop clusters. Both the Scarlett and DARE schemes perform file replications based on the popularity of files within the cluster. The Scarlett adopts a proactive approach that periodically replicates files based on its predicted popularity. Conversely, DARE proposed a reactive approach, which replicates files whenever there are changes to file popularity. Unlike Scarlett and DARE, the CDRM scheme improves the availability of files through adjustments to the number of replicas and replicas placement. Through these adjustments, CDRM will be able to dynamically distribute workload among the nodes in the cluster.

Scheduler	**Soft/Hard Deadline**	**Tracks Historical Run Times**
PD Scheduler	Soft	Yes
ARIA	Soft	No
MRCP	Soft	No
MIMP	Soft	No

Table 5 Deadline schedulers

Finally, the conflict between achieving fairness and data locality is discussed. The delay scheduler is described. Delay scheduling favors data locality over fairness albeit for a short period of time. Instead of favoring fairness or data locality, the LSAP-Fair scheduler allows system administrators to achieve a balance between the two factors.

Chapter 4. Hadoop Job Scheduling with Dynamic Task Splitting

This chapter discusses how the proposed scheduler is able to perform tradeoffs between fairness and performance by dynamically splitting a task.

4.1 Dynamic Task Splitting

The Dynamic Task Splitting Scheduler (DTSS) is a generalization of delay scheduling. It explores the tradeoff between fairness and performance (data locality) through dynamic splitting of tasks. Unlike delay scheduling where the whole task is delayed when it cannot be launched locally, dynamic task splitting improves fairness by allowing part of the task to launch on a non-data-local node while other part waits for a data-local node.

In delay scheduling (DS), when a job is delayed, the job loses its chance to process the entire block of data. However in the DTSS, processing of a single block of data can be dynamically divided into two portions: of which, one of the parts can be scheduled immediately to a node to satisfy the fairness criteria while the second part is delayed similar to delay scheduling.

The point (P) is the proportion of data that is to be divided for processing by each of the split tasks depending on the tradeoff preferences between fairness and performance (Figure 5).

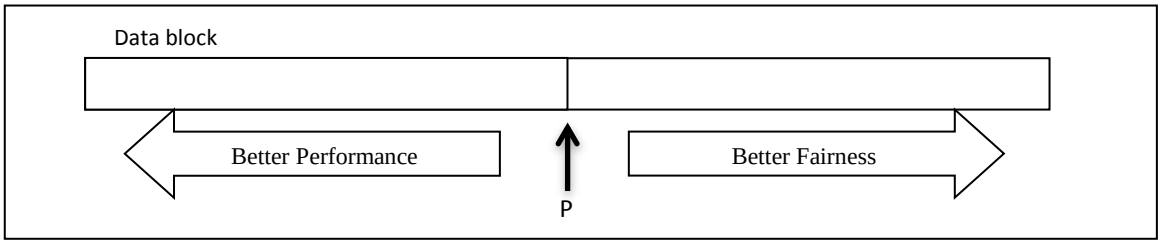

Figure 5 Tradeoff between fairness and performance

For example, given$0 \leq P \leq 1$, when P = 1, there will be 100% tradeoff in performance for fairness as the scheduler assigns the entire task to be processed at a non-data-local node. On the contrary, when P = 0, fairness will be compromised for performance where the scheduler will delay the scheduling of the task to wait for a free data-local node. This behavior is similar to delay scheduling. When a task is split, the portion of the task that is split and scheduled to the non-data-local node is known as the split task. The other part of

the task waiting for a data-local node is known as the waiting task. Analysis on the impact of the value of P will be further discussed in the next section.

4.1.1 Analysis

The DTSS ensures that users will be assigned their fair share even when there is no data-local node available thus improving fairness. However, it is also important to analyze DTSS's performance impact due to the splitting of tasks and executing part of tasks at non-data-local nodes. Executing only a portion of the original task mitigates the performance impact of executing the entire task at a non-data local node. In the event where the scheduler fails to find a data-local node for the task after timeout, the task will still be scheduled at a non-data local node.

Analysis on the scheduler will be discussed based on the following assumptions:

a) a node takes N seconds and M seconds to process a whole task at a data-local node and non-data local node respectively, where $(N < M)$. Both timings include the overheads incurred for task creation in addition to the processing time. Additionally, M includes the time to transfer the data to the non-data-local node.
b) D_{local}^{DTSS} and $D_{non-local}^{DTSS}$ are the time it takes to obtain a data-local node and non-data-local node respectively under DTSS.
c) D_{local}^{DS} and $D_{non-local}^{DS}$ are the time it takes to obtain a data-local node and non-data-local node respectively under DS.

As mentioned above, in delay scheduling, a timeout (T) will occur if a job is delayed for too long while waiting for a data-local node. As such, both $D_{non-local}^{DTSS}$ and $D_{non-local}^{DS}$ will be lower-bounded by T (10)(11):

$$D_{non-local}^{DTSS} \geq T \tag{10}$$

$$D_{non-local}^{DS} \geq T \tag{11}$$

If a job can be scheduled to a data-local node, it signifies that the timeout have not occurred. Thus, D_{local}^{DTSS} will be lower-bounded by $D_{non-local}^{DTSS}$ (12) and D_{local}^{DS}will be lower-bounded by $D_{non-local}^{DS}$ (13):

$$D_{local}^{DTSS} \leq D_{non-local}^{DTSS} \tag{12}$$

$$D_{local}^{DS} \leq D_{non-local}^{DS}. \quad (13)$$

4.1.1.1 Delay Scheduling

Under delay scheduling, the completion time for a task scheduled at a data-local node consists of only the time it takes to process the task (14):

$$T_{DS_{Local}} = N \quad (14)$$

If the task is able to secure a data-local node after waiting for D_{local}^{DS} seconds, its completion time depends on the time it takes to run the task and the time it takes to wait for the slot (15):

$$T_{DS_{DelayedLocal}} = N + D_{local}^{DS} \quad (15)$$

Lastly, if the task is unable to secure a data-local node after its timeout, its completion time will be made up of the time it takes to run the task at the non-data-local node and the time it takes to acquire the non-data-local node (16):

$$T_{DS_{NonLocal}} = M + D_{non-local}^{DS} \quad (16)$$

4.1.1.2 DTSS

Under DTSS, the time it takes to complete processing a task at a data-local node is equal to $T_{DS_{local}}$ when a data-local node for the task is available at the time of scheduling. The reason is that DTSS will always attempt to schedule a task at a data-local node whenever possible.

Assuming P where $0 \leq P \leq 1$ is the proportion of the task to be split from the original task, the completion time of a task under DTSS will depend on which half of the task is completed first.

The completion time of the split task is made up of the time to process part of the original task at the non-data-local node (17). There is no waiting time for the split task as it is scheduled immediately after the original task is selected for splitting.

$$T_{DTSS_{Split\_Task}} = (P * M) \quad (17)$$

The time to complete processing the rest of the task depends on whether the waiting task gets a data-local node. If the waiting task gets a data-local node, the time it will complete its processing will be made up of the time it takes to process the remaining data not scheduled and the waiting time before acquiring a data-local node (18):

$$T_{DTSS_{Waiting\_Local}} = ((1 - P) * N) + D_{local}^{DTSS} \quad (18)$$

If the waiting task timeout and is scheduled at non-data-local node, the time it will complete its processing will be made up of the time it takes to process the remaining data at a non-data-local node and the delay in acquiring the non-data-local node (19):

$$T_{DTSS_{Waiting\_NonLocal}} = ((1 - P) * M) + D_{non-local}^{DTSS} \quad (19)$$

Given that $N < M$ and $D_{local}^{DTSS} < D_{non-local}^{DTSS}$, the best-case scenario for DTSS when splitting is required is when waiting task gets to run locally (20):

$$T_{DTSS_{Best-case}} = \max(T_{DTSS_{Split\_Task}}, T_{DTSS_{Waiting\_Local}}) \quad (20)$$

Likewise, the worst-case scenario for DTSS when splitting is required is when the other half of the split task did not get to run locally (21):

$$T_{DTSS_{Worst-case}} = \max(T_{DTSS_{Split\_Task}}, T_{DTSS_{Waiting\_NonLocal}}) \quad (21)$$

4.1.1.3 Comparison

The performance of DTSS will be equivalent to DS if a data-local node can be secured for a task at the time of scheduling. The situation is more complicated when a data-local node cannot be secured for the task. A few scenarios will be discussed and compared in this section.

a) <u>Data-local node obtained for both DTSS and DS</u>

Assuming that the task under DTSS and DS is able to secure a data-local node after waiting for D_{local}^{DTSS} and D_{local}^{DS} seconds respectively.

Case A - $T_{DTSS} = T_{DTSS_{Split\_Task}}$

Splitting of the task will be better if the time it takes to process the part of the task at a non-data local node is lesser than the time it takes to wait for a slot (D_{local}^{DS}) and the time to process the entire task (24):

$$T_{DTSS_{Split\_Task}} < D_{local}^{DS} + N \tag{22}$$

$$P * M < D_{local}^{DS} + N \tag{23}$$

$$P < \frac{D_{local}^{DS} + N}{M} \tag{24}$$

Based on (24), the amount of time where a task has to wait for its slot to process is crucial to the performance of DTSS at different levels of splitting. When the waiting time for the data-local (D_{local}^{DS}) node is too small, the splitting point (P) will depends on the speed difference and the overheads incurred between processing on a non-data-local node and on a data-local node (25). If the N<<M, then P will tend to 0.

$$(D_{local}^{DS} \to 0) \Rightarrow (P < \frac{N}{M}) \tag{25}$$

Case B - $T_{DTSS} = T_{DTSS_{Waiting\_Local}}$

Splitting of the task will result in better performance if the waiting time for a data-local node under DTSS plus the time it takes to process the waiting task is shorter than the time it takes to process the task under DS (26):

$$T_{DTSS_{Waiting\_Local}} < D_{local}^{DS} + N \tag{26}$$

$$D_{local}^{DTSS} + (1 - P) * M < D_{local}^{DS} + N \tag{27}$$

$$P > \frac{D_{local}^{DTSS} - D_{local}^{DS}}{N} \tag{28}$$

Based on (28), if the both delays are the same, then splitting of the task will result in better performance as long as P is larger than 0.

b) <u>Non-Data-local node obtained for both DTSS and DS</u>

Assuming that the task under DTSS and DS is not able to secure a data-local node after waiting for $D_{non-local}^{DTSS}$and $D_{non-local}^{DS}$ seconds respectively.

Case A - $T_{DTSS} = T_{DTSS_{Split\_Task}}$

Splitting of the task will be better if the time to process the split task under DTSS is less than the time to process the entire data under DS (29):

$$T_{DTSS_{Split\_Task}} < D_{non-local}^{DS} + M \tag{29}$$

$$P * M < D_{non-local}^{DS} + M \tag{30}$$

$$P < \frac{D_{non-local}^{DS} + M}{M} \tag{31}$$

Based on (31), splitting will always get better performance for any $0 \leq P \leq 1$ as the delay for DS to obtain a non-data-local node is bounded by T (See (11)) and that $T \geq 0$.

Case B - $T_{DTSS} = T_{DTSS_{Waiting\_NonLocal}}$

Splitting of the task will result in better performance if the waiting time for a non-data-local node under DTSS plus the time it takes to process the waiting task is shorter than the time it takes to process the task under DS (34):

$$T_{DTSS_{Waiting\_NonLocal}} < D_{non-local}^{DS} + M \tag{32}$$

$$D_{non-local}^{DTSS} + (1 - P) * M < D_{non-local}^{DS} + M \tag{33}$$

$$P > \frac{D_{non-local}^{DS} - D_{non-local}^{DTSS}}{M} \tag{34}$$

Based on (34), if both DS and DTSS take approximately the same time to be scheduled to a non-data-local node, i.e. $D_{non-local}^{DS} = D_{non-local}^{DTSS}$, then splitting will always be better for any $0 \leq P \leq 1$.

c) <u>Data-local node for DTSS versus Non-Data-local node for DS</u>

Case A - $T_{DTSS} = T_{DTSS_{Split\_Task}}$

Please refer to (31).

Case B - $T_{DTSS} = T_{DTSS_{Waiting\_Local}}$

DTSS will outperform DS if the time to process the waiting task at a data-local node under DTSS is less than the time to process the entire data at a non-data-local node under DS.

$$T_{DTSS_{Waiting\_Local}} < D^{DS}_{non-local} + M$$

$$D^{DTSS}_{local} + (1 - P) * N < D^{DS}_{non-local} + M$$

$$P > \frac{(D^{DTSS}_{local} - D^{DS}_{non-local}) + (N - M)}{N} \quad (35)$$

Based on (35), if splitting the task takes lesser or approximately the same time to obtain a data-local node as compared to the time required by DS to obtain a non-data-local node, i.e. $D^{DTSS}_{local} = D^{DS}_{non-local}$, then splitting will always be better for any $0 \leq P \leq 1$.

The time required to process the task on a data-local node is shorter than the time required to process the data on a non-data-local node (N<M). As such, it acts as a tolerance if the time taken by DTSS to obtain a data-local node is longer than the time required by DS to obtain a non-data-local node, i.e. $D^{DTSS}_{local} > D^{DS}_{non-local}$.

d) <u>Non-Data-local node for DTSS versus Data-local node for DS</u>

This case represents the worst-case scenario for DTSS as DS is scheduled to a data-local node where the processing will be faster.

Case A - $T_{DTSS} = T_{DTSS_{Split\_Task}}$

Please refer to (24).

Case B - $T_{DTSS} = T_{DTSS_{Waiting\_NonLocal}}$

$$T_{DTSS_{waiting\_NonLocal}} < D^{DS}_{local} + N \quad (36)$$

$$D^{DTSS}_{non-local} + (1 - P) * M < D^{DS}_{local} + N \quad (37)$$

$$P * M > (D^{DTSS}_{non-local} - D^{DS}_{local}) + (M - N) \quad (38)$$

$$P > \frac{(D^{DTSS}_{non-local} - D^{DS}_{local}) + (M - N)}{M} \quad (39)$$

Based on (38), splitting the task will be better if the time to process the split task ($P * M$) is more than the additional time to wait for a non-data-local node($D^{DTSS}_{non-local} - D^{DS}_{local}$) and the additional time to process the data at that node ($M - N$).

4.1.2 Pseudo code

The pseudo code for DTSS is presented in Algorithm 1**.** DTSS is very similar to delay scheduling in that it adopts the mechanism for delaying tasks for jobs when it is not possible to split tasks anymore. When P=0, the scheduler will operate like a delay scheduler (line 6). DTSS will always attempt to schedule tasks to data-local nodes whenever it is possible (line 10). If no data-local nodes are available, DTSS will first check if a timeout had occurred for the job, if yes, it would schedule the entire task at a non-data-local node (line 12-21).

Algorithm 1 *Dynamic Task Splitting Scheduler*

Input:
TaskTrackers $TS_0, TS_1 \ldots TS_N$ *in the Hadoop cluster*
Jobs $j_0, j_1 \ldots j_W$
Timeout value where a non-data-local node can be scheduled, T_1
Proportion of task to be split, P
Output:
Task to be executed

```
1.  If a heartbeat is received from TS_N then
2.      For each job j with skipped = true, increase j.wait by the time difference since last
3.      heartbeat and set j.skipped = false
4.      If TS_N has a free map slot then
5.          For next job j in jobs then // the order of jobs is determined by the Fair Scheduler
6.              If P == 0 then // no splitting
7.                  Task T = DelayScheduling (TS_N, T_1) // initiate delay scheduling
8.                  Return T
9.              Else
10.                 Task T = j.obtainNodeLocalMapTask()
11.                 If T is not found then
12.                     If j.level = 1 or j.wait ≥ T_1 then
13.                         // Timeout occurred or job was previously scheduled
14.                         // non-data-local node
15.                         T=j.getNonLocalMapTask() //obtain a non-local task
16.                         If T is found then
17.                             Set j.wait=0,j.level=1
18.                             Return T
19.                         Else
20.                             set j.skipped = true
21.                         Endif
22.                     Endif
23.                     Else
24.                         Task T_RL = j.findNonLocalMapTask ()
25.                         If T_RL is found then
26.                             T_RL_split = CreateDynamicSplitTask (T_RL, P)
27.                             Set j.level=0
28.                             Return T_RL_split
29.                 Else
30.                     Set j.wait=0, j.level=0
31.                 Return T
32.             Endif
33.     Endif
34. Endif
```

Task splitting will happen after the above checks are done. DTSS first searches for a task that is not local to the tasktracker and is not split before to prevent multiple splitting of a task (line 11). If a task is found, the DTSS will create a dynamic task and schedule the newly created task (line 13). When a dynamic task is scheduled, parameter *wait* will not be reset, unlike in delay scheduling where *wait* will be reset when a task is scheduled.

Starvation is avoided by setting a maximum delay parameter T_1. Each job has a parameter known as wait, that tracks how long a job have been waiting for a data-local node and a parameter known as skipped which tracks if a job have been skipped for scheduling previously. Additionally, jobs also have a parameter known as level that tracks if a job should be scheduled to a non-data-local node. Similar to delay scheduling, all jobs starts with level=0 which signifies that the jobs can only be scheduled on a data-local node. Jobs can only be scheduled at a non-data-local node if they are skipped longer than the stipulated delay T_1. Once a job exceeds the waiting time, its level will be set to 1 and the job will be scheduled to a non-data-local node unless it is able to obtain a data-local node.

4.2 Implementation of the DTSS

As mentioned in Chapter 2, each map task in Hadoop is designed to work on an entire data block and the number of map tasks in a job is determined in job submission. Hence, modifications to the Hadoop architecture are required in order to support DTSS. This section will discuss the work done on the implementation of the scheduler and the modifications done on the Hadoop framework to support it.

4.2.1 Integrating DTSS

The Hadoop software architecture is designed to support the addition of custom schedulers. Custom schedulers are required to extend the *TaskScheduler* class in order to be recognized (Figure 6).

Once integrated, the scheduler can be selected by setting the *mapred.jobtracker.taskscheduler* parameter in the Hadoop configuration file. Doing so will instruct Hadoop to initialize the appropriate scheduler and refer to it for scheduling decisions when a map or reduce slot is available.

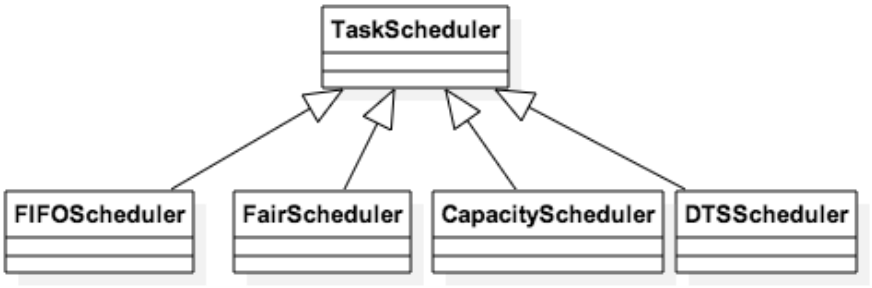

Figure 6 Schedulers extending TaskScheduler class

Like the delay scheduler, DTSS is built into the Fair Scheduler and relies on it to enforce the minimum share for each user. The Fair Scheduler performs the first level of scheduling by deciding on the priority of the users to schedule while the DTSS performs the second level of scheduling for the tasks assignment and launching.

Under the Fair Scheduler, the *PoolSchedulable class* is responsible for selection of jobs within the pool for execution while the *JobSchedulable class* is responsible for the selection and launching of tasks based on delay scheduling. A *PoolSchedulable* object manages a set of *JobSchedulable* objects.

Both the PoolSchedulable class and JobSchedulable class are sub-classes of the *Schedulable class. The Schedulable* class represents an entity that can launch tasks. The class diagram when delay scheduling is used is shown in Figure 7.

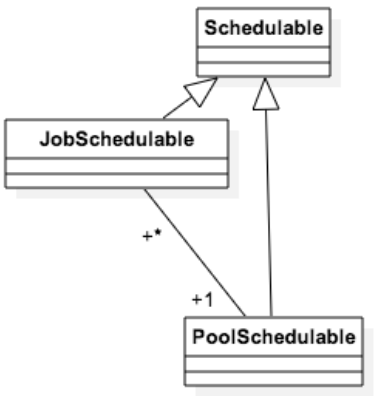

Figure 7 Class diagram when delay scheduling is used

DTSS is integrated into Fair Scheduler by creating the DTSSJobSchedulable class, which inherits the Schedulable class and redirecting calls to this class (Figure 8).

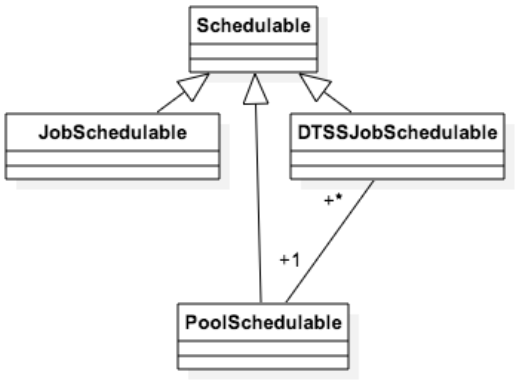

Figure 8 Class diagram after integrating DTSS

4.2.2 Task Splitting

The splitting of a task consists of the following steps:

1. Computing the portion of data that a task and its split task is responsible for after splitting
2. Creating the split task, assigning it to its job and its execution
3. Ensuring that the reduce tasks do not stop execution after all map tasks are completed

4.2.2.1 Computing offsets and data length for split task

Task splitting requires that the scheduler define the point of splitting (P) as described in Section 4.1 . The point of splitting is used to logically divide a task's data block (Task A) into two parts; one for scheduling at a non-data-local node (Task B) while the other waits for a data-local node to be ready (Task C) (Figure 9). Each task will be allocated its portion of the data defined by a starting offset and the length of data to process.

Four levels of splitting are currently defined in DTSS. There is no limitation on how fine the task split can be theoretically.

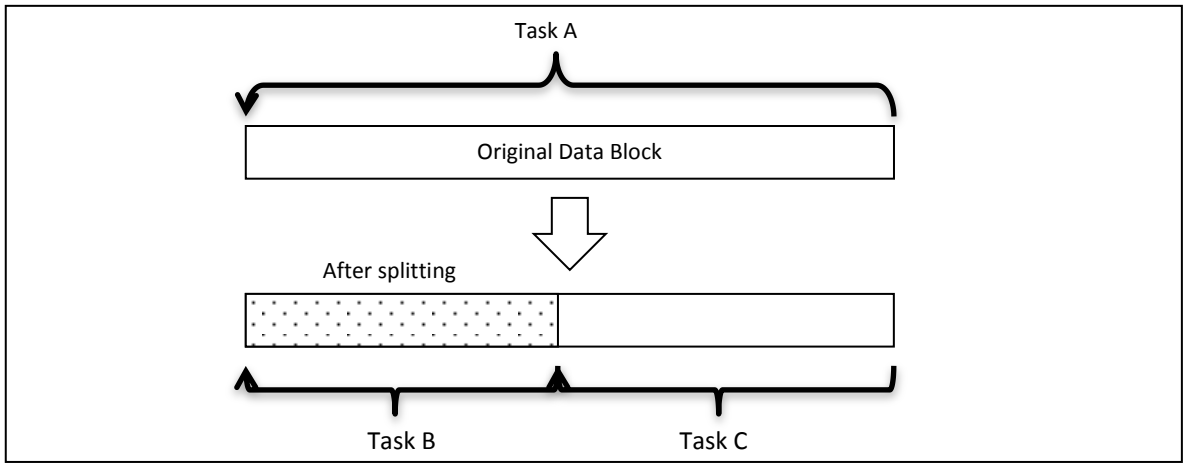

Figure 9 Task splitting example

4.2.2.2 Split task creation

A job in Hadoop handles the information regarding its tasks and their statuses. In Hadoop MapReduce, the number of tasks in a job is defined and fixed during job submission. There are no provisions for modifying the number of tasks dynamically.

A job tracks its map tasks, a setup task and a cleanup task using an array. The setup task performs setup functions during job submission while the cleanup task performs clean up functions at job completion. The setup and clean up tasks are stored at the end of the task array (Figure 10). Each map tasks are given an identifier based on its position in the array. The JobTracker monitors the job's tasks through the task's identifier. Dynamically created tasks are appended to the task array (Figure 11).

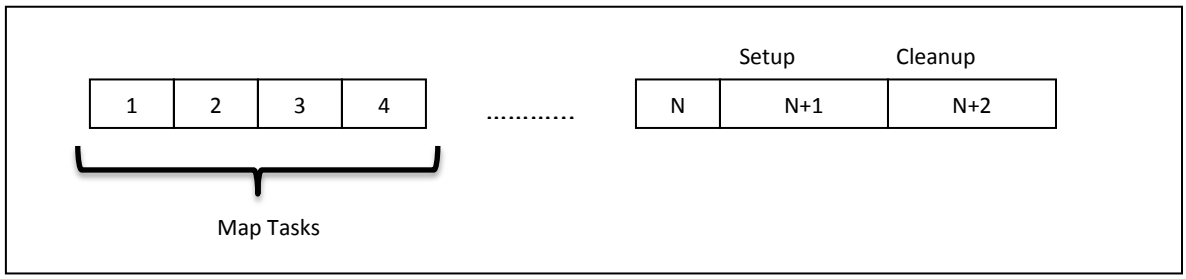

Figure 10 Map Task, Setup and Cleanup task identifiers

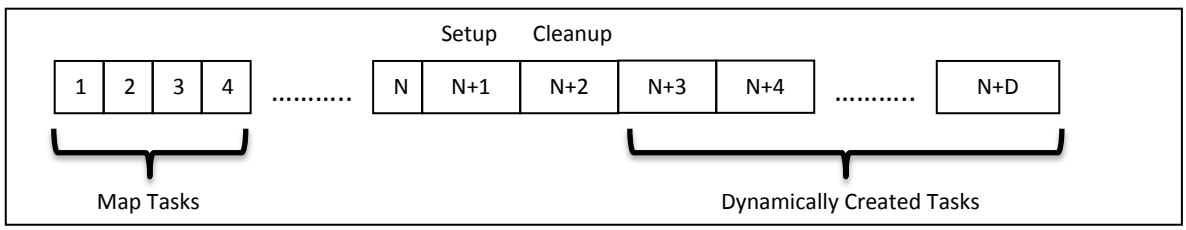

Figure 11 Identifiers for dynamic created tasks

4.2.2.3 Task assignments

Task assignments are done through the transmission of serialized java objects (*MapTask.java or ReduceTask.java*) to the remote TaskTrackers. These java objects contain information that the TaskTracker requires to create the actual process on its machine. The information included is the input file, the job that the task belongs to and other configuration details. In the original Hadoop implementation, each map tasks will process its input data in entirety. As such, to support DTSS, the scheduler will need to send the start offset and length of data to process for the task to the TaskTracker. The TaskTracker will pass on this information to the task to inform it of the portion of the data it is responsible for.

4.2.3 Informing reduce tasks that all map tasks are completed

Reduce tasks tracks the completion of map tasks for copying of the map task output. An event (*TaskCompletionEvent)* will be sent from the JobTracker to the reduce task whenever a map task completes.

Reduce tasks counts the number of tasks completion events and waits for map tasks output. It will stop waiting for map outputs once the number of task completion event reaches the number of map tasks in the job. The reduce tasks are informed about the number of map tasks in its job during the reduce tasks creation. There are no mechanism to update the number of map tasks in the reduce tasks, as it is expected to remain static in the original Hadoop implementation.

In DTSS, the number of map tasks will dynamically increase due to the splitting of the map tasks. The reduce tasks will omit the outputs from the newly created map tasks if they are not informed about the increase in the number of map tasks. As a result, the *TaskCompletionEvent* is modified to include a flag on whether all map tasks are completed. The reduce tasks are then modified to monitor the flag instead of counting the number of completed map tasks. This is possible as the *JobTracker* has a complete picture of all the jobs and their statuses.

4.3 Other Considerations

By dividing tasks across all the nodes in the cluster, Hadoop allows for nodes that are slow to delay the completion of a job. To prevent this from happening, Hadoop will schedule redundant copies of the remaining tasks when most of the tasks in a job are

completed. Whichever copy of the task completes first will be considered the definitive copy and other copies of the tasks will be terminated. This is known as speculative execution. Tasks split by DTSS can also be speculatively executed. A split task that is speculatively executed will process the same data as the original split task. DTSS also do not affect the fault-tolerance mechanism when split tasks fails in Hadoop. Split tasks that fail will re-executed.

4.4 Summary

The Dynamic Task Splitting Scheduler (DTSS) is presented in this chapter. Analysis of the DTSS and delay scheduling is also discussed. Implementation details of the DTSS and the modifications performed on the Hadoop source codes so as to support DTSS are also presented. The experiments and the results will be presented in the next chapter.

Chapter 5. Experiments and Results

This chapter describes the experiments performed using the DTSS. The experimental cluster setup will be presented. Comparisons are made against the Delay Scheduler (DS).

5.1 Design of Experiments

5.1.1 Experimental cluster setup

A cluster has been setup for the experimental purposes. The cluster is made up of five nodes interconnected through a 100Mbps network. The nodes come with the following configuration.

CPU	Intel(R) Core(TM) 2 Extreme CPU Q6850 @ 3.00GHz (4 cores)
RAM	4 GB
OS	RHEL 4.4
HDD	1 TB

One of the nodes will host the JobTracker and Namenode while the remaining four nodes will each host an instance of the TaskTracker and Datanode. Each TaskTracker is configured to host one map slot and one reduce slot. This configuration represents a resource scarce environment. It emulates a busy cluster when the experimental workloads are executed.

5.1.2 Experimental Workload

The workload used for the experiments is created using the Statistical Workload Injector for MapReduce (SWIM) framework [32]. SWIM is meant for performance measurements of MapReduce clusters. SWIM is able to analyze actual workloads and synthesize them into representative workloads that follow the schedule. It then generates MapReduce jobs based on the representative workload information.

To create the workload, a custom schedule consisting of 20 jobs with a submission rate of 30 seconds is fed into SWIM. Modifications are made to SWIM so as to randomly assign jobs in the workload to three users. Each user will submit its assigned jobs to the cluster based on the custom schedule. This is to simulate multiple users sharing the cluster simultaneously. The number of map tasks varies from five to ten map tasks. The purpose of the map and reduce tasks in these workload is to simulate the load on the Hadoop

cluster. The map tasks used in the workload are IO-bound. Each map task reads data from HDFS and writes out the same data to the reducers. The output data from the map tasks will then be sorted and shuffled to the reduce tasks. Each reduce tasks will read the output data from the map tasks and randomly writes some output to HDFS.

5.1.3 Performance Evaluation

Both the DTSS and DS are embedded within the Fair Scheduler. The Fair Scheduler performs the hierarchical scheduling capability while DTSS and DS decide whether to schedule the job selected by the Fair Scheduler.

The makespan of a user is the amount of time it takes for the user to complete running all its jobs. A shorter makespan for all users means that the cluster has better throughput. Two configurations for DTSS are used for comparisons against the DS. DTSS (P=0.25) means 75% of the task will attempt to wait for a data-local node while the rest will be scheduled to a non-data-local node immediately. Similarly, DTSS (P=0.75) means that 25% of the task will attempt to wait for a data-local node while the rest will be scheduled to a non-data-local node immediately.

Data distribution plays an important role under Hadoop as it determines how easy is it for a job to obtain a data-local node in the cluster. Two data distributions are considered for the performance evaluations: skewed distribution and equal distribution. Under the skewed data distribution configuration, all the input data for the jobs are situated in 20% of the nodes whereas under the equal data distribution configuration, data are distributed equally across all the nodes. This is to test the performance of DTSS against DS under different workload distributions.

5.2 Results and Experiments

5.2.1 Skewed Distribution

Under the skewed data distribution, it is very difficult for jobs to obtain a data-local node as only 20% of the nodes (1 node in experiment configuration) contain the data in which the jobs require. This is to emulate situations in bigger clusters where it is less likely to obtain a data-local node. For example, in a 1000-nodes cluster with a replication of three for each file under HDFS, the probability of getting a data-local node is only 3 out of the 1000 nodes.

Figure 12 shows that the makespan obtained for user0 to user2 when different schedulers are used. The makespan for DTSS (P=0.25) improves by 2% to 11%. It can also be observed that the makespan for DTSS (P=0.75) improves by approximately 1.5% for user1.

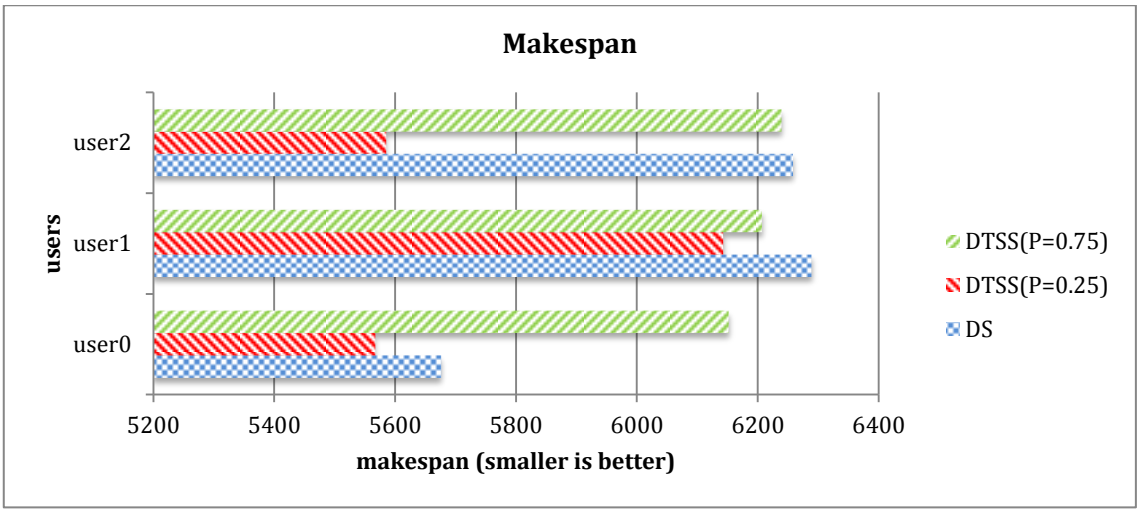

Figure 12 Makespan obtained for skewed data distribution

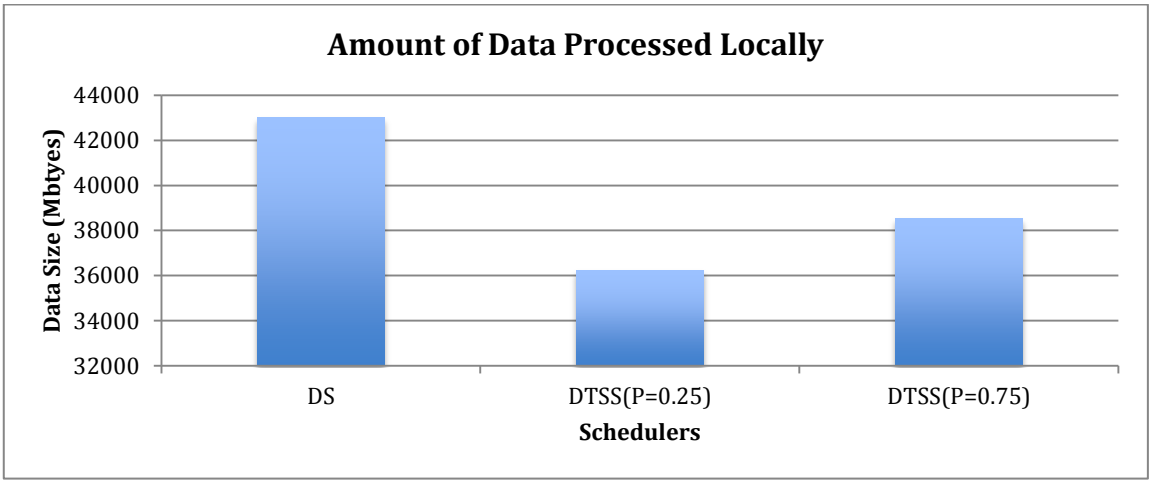

Figure 13 Amount of Data Processed Locally

Figure 13 shows the amount of data processed in data-local nodes across all users for each scheduler tested. As shown, DS processed the most amounts of data at data-local nodes compared to the DTSS. It is interesting to note that despite processing more data locally, the performance of DS is still worse than DTSS (P=0.25). DTSS performs better as the time is better spent to process part of the task first than waiting for a data-local node under DS. The savings for processing the rest of the data locally is not significant as

compared to the time saved by processing 25% of the data early and the time spent waiting for a slot (See Section 4.1.1.3).

However, the performance of DTSS will deteriorate if too much of the task is split and processed at a non-data local node. In this case, DTSS (P=0.25) appears to be a good compromise for performance. This experiment shows that the performance advantage that data locality brings might be outweigh if the time it takes to obtain a data-local slot is long.

5.2.2 Equal Distribution

In the equal data distribution configuration, data are spread equally throughout the cluster. It can be observed in Figure 14 that the performance of DTSS (P=0.25) is worst for all users as compared to DS. After further investigation, it was noted that DS is able to obtain 100% data locality for all users whereas DTSS (P=0.25) is only able to obtain at most 39% data locality.

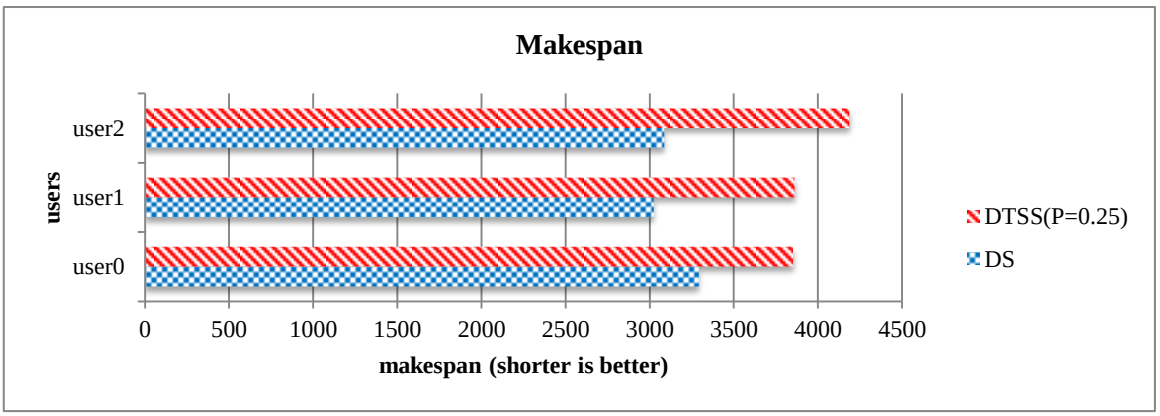

Figure 14 Makespan obtained under equal distributed data

Under an equal data distribution, it is very likely for a task to be scheduled to a data-local node. Tasks running on data-local nodes are processed faster thus slots are also released at a faster rate. This is beneficial for DS, as it will not be delayed for long before a data-local node becomes available. However, under DTSS, tasks will be split and scheduled to non-data-local nodes without delay. This may take up a slot that is data-local for other jobs especially under an equal data distributed configuration. Hence, DTSS is not a suitable scheduler where users' workload data are equally distributed across the cluster.

5.2.3 Fairness Evaluation

As mentioned previously, DS compromised the fairness of a user when it chooses to delay the user's job from executing its task when it cannot obtain a data-local node. Unlike DS, a user's fairness can be adjusted based on the splitting of a task. Figure 15 shows the map slots allocation under DS and DTSS across their entire scheduling epochs respectively. These graphs also give and overview on the sharing of map slots across different users at each scheduling epoch. The scheduling traces shown in Table 6 and Table 7 are extracted from these allocation graphs respectively.

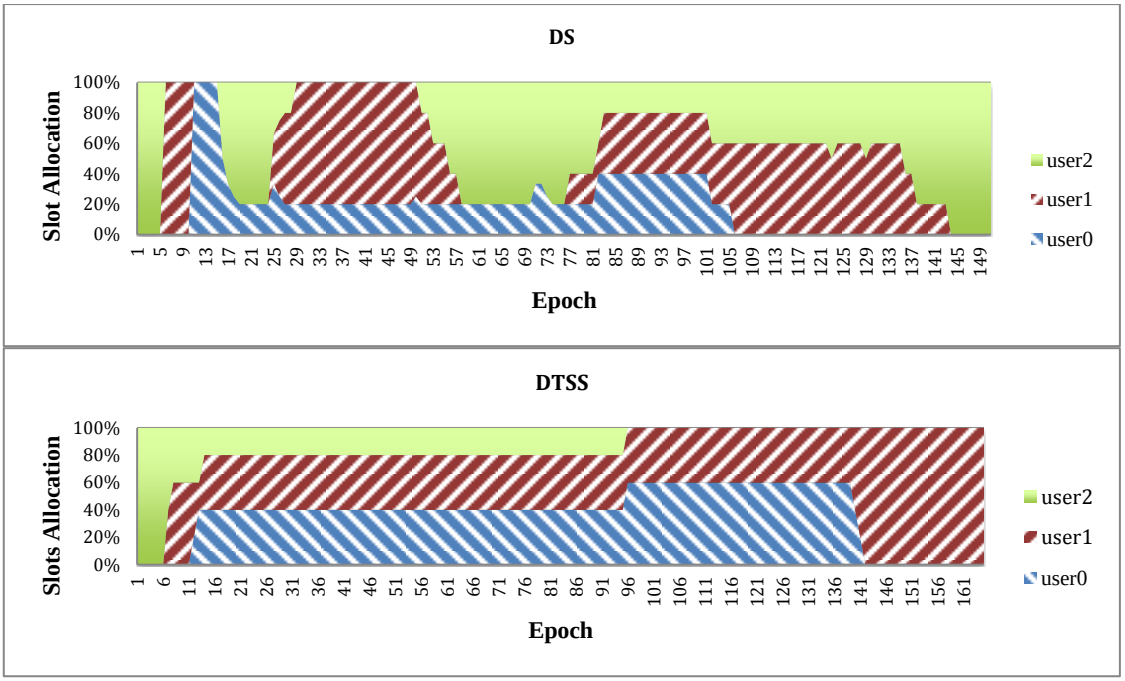

Figure 15 Map Slots allocation under DS and DTSS

Based on the allocation graph for DS, it can be seen that only user2 is granted map slots from epoch 1 -5 as it is the only user in the cluster. Likewise only user1 and user0 are allocated map slots at epoch 6 and 13 respectively. It is observed that there is no sharing of the cluster resources from epoch 1 to 20 as the scheduler did not assign map slots to all users. The sharing of cluster resources between user0 and user2 is observed from epoch 21 to 25 where user2 is assigned more resources than user0. It can be seen that user0 does not have any request for resources after epoch 106.

Looking at the allocation graph under DS as a whole, it can be observed that the cluster resources are not shared fairly between the users. In contrast, under DTSS, it can be seen that cluster resources are shared equally among the users. For example, looking at the

allocation graph for DTSS, it can be seen that all users are granted an almost equal share of the cluster resources from epoch 11 to epoch 96. The cluster resources are then shared equally between user0 and user1 from epoch 97 to epoch 141 as user2 do not have any request for resources after epoch 96.

Next, the first 20 scheduling epochs for both DS and DTSS are described. The expected share of a user refers to the fair share a user should obtain when sharing the cluster with other users. The expected fair share of a user i is obtained by dividing the number of map slots by the number of users (40).

$$User_i^{expected\_share} = \frac{\#\ map\ slots}{\#\ of\ users} \tag{40}$$

A fair share ratio is defined to measure how well a user is able to obtain its fair share in the cluster. The fair share ratio at each scheduling epoch is computed by dividing a user's current share by its expected share (41). A fair share ratio that is equal one means that the user is able to obtain its fair share. A value of more than one means that the user is allocated more than its fair share. Lastly, a value of less than one means that the user is assigned resources below its fair share.

$$User_i^{fair\_share\_ratio} = \frac{User_i^{current\_share}}{User_i^{expected\_share}} \tag{41}$$

DS: Slots Assignments				
EPOCH	**user0**	**user1**	**user2**	**Expected Share**
1	NA	NA	1	5
2	NA	NA	1	5
3	NA	NA	1	5
4	NA	NA	1	5
5	NA	NA	1	5
6	NA	1	0	2.5
7	NA	1	0	2.5
8	NA	1	0	2.5
9	NA	1	0	2.5
10	NA	1	0	2.5
11	1	0	0	1.67
12	1	0	0	1.67
13	1	0	0	1.67
14	1	0	0	1.67
15	1	0	0	1.67
16	1	0	1	1.67
17	1	0	2	1.67
18	1	0	3	1.67
19	1	0	4	1.67
20	1	0	4	1.67

Table 6 First 20 scheduling epochs under DS

DTSS: Slots Assignments				
EPOCH	**user0**	**user1**	**user2**	**Expected Share**
1	NA	NA	1	5
2	NA	NA	2	5
3	NA	NA	3	5
4	NA	NA	4	5
5	NA	NA	5	5
7	NA	1	4	2.5
7	NA	2	3	2.5
8	NA	3	2	2.5
9	NA	3	2	2.5
10	NA	3	2	2.5
11	0	3	2	1.67
12	1	2	2	1.67
13	2	1	2	1.67
14	2	2	1	1.67
15	2	2	1	1.67
16	2	2	1	1.67
17	2	2	1	1.67
18	2	2	1	1.67
19	2	2	1	1.67
20	2	2	1	1.67

Table 7 First 20 scheduling epochs under DTSS

Table 6 and Table 7 show the first 20 scheduling epochs under DS and DTSS respectively. In each table, "NA" implies that the user has not submitted any job to the cluster at that scheduling epoch. The numbers under each user represent the number of map slots assigned to the user. The expected share for each user is shown under the corresponding column. For example, at epoch 1, only user2 had submitted a job to the cluster. Hence, with the experimental cluster of five nodes, the expected share for user2 is five. Likewise, at epoch 6, user1 submitted a job to the cluster, thus the expected share for each user is now 2.5.

Table 8 and Table 9 show the fair share ratio obtained for each user at each scheduling epoch under DS and DTSS respectively. It can be seen that the fair share ratio for each user is low under DS especially during the beginning as the user's job are being delayed. Conversely, the fair share ratio for each user is much better in DTSS throughout the entire trace.

Through analysis of the DS scheduling trace in Table 6, it can be observed that from epoch 1 to epoch 5, only one map slot is assigned to user2 when its expected share is five. This translates to a fair share ratio of 0.2 as shown in Table 8. The reason why user2 is not able to achieve its expected share is due the delay imposed by DS, on the user's job when it is not able to obtain data-local nodes for its tasks. This phenomenal can be

observed in the first 15 scheduling epochs. Contrast with the DTSS scheduling trace in Table 7, it can be seen that at user2 is progressively assigned up to 5 map slots from epoch 1 to epoch 5. This translates to a fair share ratio of one as shown in Table 9. The reason why user2 is assigned with five map slots is due to the splitting of task when the user's job is not able to obtain data-local nodes for its tasks.

DS: Fair Share Ratio			
Epoch	**user0**	**user1**	**user2**
1	NA	NA	0.2
2	NA	NA	0.2
3	NA	NA	0.2
4	NA	NA	0.2
5	NA	NA	0.2
6	NA	0.4	0
7	NA	0.4	0
8	NA	0.4	0
9	NA	0.4	0
10	NA	0.4	0
11	0.6	0	0
12	0.6	0	0
13	0.6	0	0
14	0.6	0	0
15	0.6	0	0
16	0.6	0	0.6
17	0.6	0	1.2
18	0.6	0	1.8
19	0.6	0	2.4
20	0.6	0	2.4

Table 8 Fair Share Ratio per scheduling epoch under DS

DTSS: Fair Share Ratio			
Epoch	**user0**	**user1**	**user2**
1	NA	NA	0.2
2	NA	NA	0.4
3	NA	NA	0.6
4	NA	NA	0.8
5	NA	NA	1
6	NA	0.4	1.6
7	NA	0.8	1.2
8	NA	1.2	0.8
9	NA	1.2	0.8
10	NA	1.2	0.8
11	0.0	1.8	1.2
12	0.6	1.2	1.2
13	1.2	0.6	1.2
14	1.2	1.2	0.6
15	1.2	1.2	0.6
16	1.2	1.2	0.6
17	1.2	1.2	0.6
18	1.2	1.2	0.6
19	1.2	1.2	0.6
20	1.2	1.2	0.6

Table 9 Fair Share Ratio per scheduling epoch under DTSS

The amount of fairness under DTSS can be tweaked by considering the percentage of a task that is split. The higher the percentage of the task is split, the better the fairness. Ie. Under the same condition, scheduling a 75% split is deemed to be fairer than scheduling a 25% split as the user gets to process higher amount of the task immediately. However, regardless of the split, DTSS will still have better fairness compared to DS as shown above.

5.3 Summary

This chapter describes the experimental setup and the workload used for the experiments. The experiments performed and the results observed are also analyzed and presented. It is shown that DTSS is able to out-perform DS under a skewed data distribution configuration. The reasons as to why DTSS's performance is worse than DS under an equal data distribution configuration is also discussed. Lastly, the ability of DTSS to obtain better fairness compared to DS is also presented.

The number of nodes as well as the capabilities of the nodes in the experimental setup limits the experiments conducted. With only five compute nodes, it is difficult to truly examine the behavior of the schedulers realistically. For example, with a bigger cluster, the chances of getting a data-local node will be much lesser compared to small clusters such as the experimental setup.

One aspect of DTSS that requires further analysis is the impact of overheads incurred on other jobs due to the splitting of tasks. Through splitting of tasks, DTSS is able to improve the fairness, as a task for a user will definitely be scheduled when a slot is available and the Fair Scheduler determines that the jobs from the user should be scheduled. However, by splitting and assigning the dynamically split task, it also increases the number of tasks for the job. This may increase the amount of time waiting for slots by other jobs, as the jobs with dynamically split tasks now requires more scheduling cycles to be assigned. I.e. The competition for slots among the slots now increases.

Another aspect worth investigating is the minimal amount of split for tasks of different size. If a task is small, splitting a task may incur a significant amount of overhead over the actual processing. For example, if a split task only takes 20 seconds to execute and the

overhead of creating the split task is 10 seconds, then the overhead for the split task is 50% of the entire task execution time, which is highly inefficient.

The next chapter will conclude the thesis and discuss potential future work to be done.

Chapter 6. Conclusion and Future Work

6.1 Conclusion

Job execution efficiency is a crucial component aspect in Hadoop as it impacts the overall throughput of a Hadoop cluster. An overview of Hadoop is covered in Chapter 2. This thesis also covers the state of the art for scheduling in Hadoop in Chapter 3. The Dynamic Task Splitting Scheduler (DTSS) is introduced in Chapter 4 followed by the presentation and analysis of experimental results in Chapter 5.

Improving the data locality of tasks improves cluster performance as it reduces the amount of overhead through reduction in the number of data transfers to other nodes for processing. In a Hadoop cluster shared by multiple users, it is important to ensure that users do not hog the cluster resources. Hence it is important to be fair to all users in terms of the sharing of the cluster resources. However, there exist a conflict between data locality and fairness. For example, when a user's job needs to be scheduled and there is no data-local node available. The scheduler may choose to ignore data locality and schedule the task on a non-data-local node. This compromises performance, as the task from the job needs to copy the data from one node to another for processing. Conversely, fairness will be compromised if the scheduler, like the Delay Scheduler (DS), chooses to forgo its slot and wait for a data-local node.

DTSS seeks to tradeoff between fairness and data locality through dynamic splitting of tasks. DTSS is able to dynamically split a task and schedule one part of the original tasks for processing at a non-data local node immediately. This approach improves fairness while only sacrificing part of the data locality for the split task. This approach is unlike the DS where fairness is entirely compromised when a task cannot be scheduled to a data-local node.

DTSS is shown to improve the makespan of users in the cluster by 2% to 11% as compared to DS under a skewed data distribution configuration even when DS is shown to achieve better data locality. The results show it is possible to improve the performance and the fairness of users through dynamic splitting of tasks. It is also shown that DS will outperform DTSS when it is easy to obtain a data-local node.

6.2 Future Work

Although DTSS is able to improve the overall makespan, it is still not possible to dynamically determine optimal splitting point for different workloads. As mentioned above, DTSS performance is reliant on the amount of time required to process a data block and the amount of time required to obtain a processing node. Hence, work can be done to monitor the amount of time required to process a data block for different workloads and to estimate the time to acquire a node. With this information, DTSS can dynamically adjust the splitting point or to make the decision if it is worthwhile to split a task, so as to obtain even better performances.

Further analysis on the impact of overheads incurred on other jobs due to the splitting of tasks can also be done. Splitting and assigning of dynamically split tasks, increases the number of tasks for the job. This may in-turn increase the amount of time waiting for slots by other jobs, as the jobs with dynamically split tasks now requires more scheduling cycles to be assigned.

Another possible extension of DTSS is to consider dynamic replication creation schemes like Scarlet [27] and DARE [28]. Higher data replications improve the chances of data locality as more nodes contain the data required by jobs. Dynamic replication schemes attempts to improve data locality by dynamically creating more replicas for popular files. Under DTSS, the data for the split tasks are copied to the non-data-local node for processing and is discarded after processing. Instead of discarding the copied data, DTSS can choose to save the copied data and use it as a replica for future scheduling. This acts as a form of dynamic replication as it increases the number of replicas for the input data and improves the chance for data locality in the future. As the number of replicas increase, the splitting of tasks for the replicated data can then scaled down as splitting may be detrimental for performance as shown in Section 5.2.2 .

Many large corporations such as Yahoo, Google and Microsoft own data centers around the world. These data centers are significant consumers of energy due to their computing equipment and cooling facilities. As Hadoop clusters typically exist within these data centers, is also interesting to investigate the energy consumption resulting from the use of DTSS.

References

[1] D. Laney. (2012, 2015). *Deja VVVu: Others Claiming Gartner's Construct for Big Data*. Available: http://blogs.gartner.com/doug-laney/deja-vvvue-others-claiming-gartners-volume-velocity-variety-construct-for-big-data/

[2] M. Isard, *et al.*, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS Operating Systems Review*, 2007, pp. 59-72.

[3] T. White, *Hadoop: The Definitive Guide, Second Edition*: O'Reilly Media Inc. , 2010.

[4] V. K. Vavilapalli, *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013, p. 5.

[5] M. Zaharia, *et al.*, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, pp. 10-10.

[6] S. Leo. (2012). *Hadoop Wiki - PoweredBy*. Available: http://wiki.apache.org/hadoop/PoweredBy

[7] A. Verma, *et al.*, "ARIA: automatic resource inference and allocation for mapreduce environments," in *Proceedings of the 8th ACM international conference on Autonomic computing*, 2011, pp. 235-244.

[8] (2012). *Capacity Scheduler Guide*. Available: http://hadoop.apache.org/docs/stable/capacity_scheduler.html

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

[10] T. Sandholm and K. Lai, "Dynamic proportional share scheduling in hadoop," in *Job scheduling strategies for parallel processing*, 2010, pp. 110-131.

[11] X. Zhang, *et al.*, "An effective data locality aware task scheduling method for MapReduce framework in heterogeneous environments," in *Cloud and Service Computing (CSC), 2011 International Conference on*, 2011, pp. 235-242.

[12] Y. Luo and B. Plale, "Hierarchical MapReduce Programming Model and Scheduling Algorithms," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, 2012, pp. 769-774.

[13] (2012). *HOD Scheduler*. Available: http://hadoop.apache.org/docs/r0.21.0/hod_scheduler.pdf

[14] M. Zaharia, *et al.*, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008, pp. 29-42.

[15] M. Zaharia, *et al.*, "Job scheduling for multi-user mapreduce clusters," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-55*, 2009.

[16] M. Hammoud and M. F. Sakr, "Locality-Aware Reduce Task Scheduling for MapReduce," in *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, 2011, pp. 570-576.

[17] J. Polo, *et al.*, "Performance-driven task co-scheduling for mapreduce environments," in *Network Operations and Management Symposium (NOMS), 2010 IEEE*, 2010, pp. 373-380.

[18] A. Verma, *et al.*, "Two Sides of a Coin: Optimizing the Schedule of MapReduce Jobs to Minimize Their Makespan and Improve Cluster Performance," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, 2012, pp. 11-18.

[19] (2013). *Apache Hadoop Module 1: Tutorial Introduction*. Available: http://developer.yahoo.com/hadoop/tutorial/module1.html

[20] A. Rasooli and D. G. Down, "COSHH: A classification and optimization based scheduler for heterogeneous Hadoop systems," *Future Generation Computer Systems*, vol. 36, pp. 1-15, 2014.

[21] J. A. Hartigan and M. A. Wong, "Algorithm AS 136: A k-means clustering algorithm," *Applied statistics*, pp. 100-108, 1979.

[22] Y. Yao, *et al.*, "Scheduling heterogeneous MapReduce jobs for efficiency improvement in enterprise clusters," in *International Symposium on Integrated Network Management (IM 2013)*, 2013, pp. 872-875.

[23] S. M. Johnson, "Optimal two-and three-stage production schedules with setup times included," *Naval research logistics quarterly*, vol. 1, pp. 61-68, 2006.

[24] A. Verma, *et al.*, "Play It Again, SimMR!," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, 2011, pp. 253-261.

[25] N. Lim, *et al.*, "A Constraint Programming Based Hadoop Scheduler for Handling MapReduce Jobs with Deadlines on Clouds," presented at t*he Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, Austin, Texas, USA, 2015.

[26] W. Zhang, *et al.*, "MIMP: deadline and interference aware scheduling of Hadoop virtual machines," in *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid),*, 2014, pp. 394-403.

[27] G. Ananthanarayanan, *et al.*, "Scarlett: coping with skewed content popularity in mapreduce clusters," in *Proceedings of the sixth conference on Computer systems*, 2011, pp. 287-300.

[28] C. L. Abad, *et al.*, "DARE: Adaptive data replication for efficient cluster scheduling," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, 2011, pp. 159-168.

[29] Q. Wei, *et al.*, "CDRM: A cost-effective dynamic replication management scheme for cloud storage cluster," in *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, 2010, pp. 188-196.

[30] Z. Guo, *et al.*, "Investigation of data locality in MapReduce," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, 2012, pp. 419-426.

[31] Z. Guo, *et al.*, "Investigation of data locality and fairness in MapReduce," in *Proceedings of third international workshop on MapReduce and its Applications Date*, 2012, pp. 25-32.

[32] Y. Chen, *et al.*, "The case for evaluating MapReduce performance using workload suites," in *19th International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2011, pp. 390-399.