Summer 6-10-2019

Scheduling and Prefetching in Hadoop with Block Access Pattern Awareness and Global Memory Sharing with Load Balancing Scheme

Sai Suman
University of Nebraska - Lincoln, saisuman@huskers.unl.edu

SCHEDULING AND PREFETCHING IN HADOOP WITH BLOCK ACCESS PATTERN AWARENESS AND GLOBAL MEMORY SHARING WITH LOAD BALANCING SCHEME

by

Sai Suman

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Ying Lu

Lincoln, Nebraska

March, 2019

SCHEDULING AND PREFETCHING IN HADOOP WITH BLOCK ACCESS PATTERN AWARENESS AND GLOBAL MEMORY SHARING WITH LOAD BALANCING SCHEME

Sai Suman, M. S.

University of Nebraska, 2019

Adviser: Ying Lu

Although several scheduling and prefetching algorithms have been proposed to improve data locality in Hadoop, there has not been much research to increase cluster performance by targeting the issue of data locality while considering the 1) cluster memory, 2) data access patterns and 3) real-time scheduling issues together.

Firstly, considering the data access patterns is crucial because the computation might access some portion of the data in the cluster only once while the rest could be accessed multiple times. Blindly retaining data in memory might eventually lead to ineffcient memory utilization.

Secondly, several studies found that the cluster memory goes highly underutilized, leaving much room that can be leveraged for storing input data for future tasks. Leveraging the aforementioned memory underutilization in the clusters is important since the nodes are usually equipped with large amounts of memory.

Thirdly, enabling a prefetching mechanism to retain popular blocks in memory could eventually lead to memory shortage, we thus present two cache eviction algorithms to evict the data that will not be accessed frequently. Furthermore, the caching mechanism could potentially lead to unbalanced utilization of memory in a cluster's nodes, so we present a mechanism for balancing the memory loads across the cluster such that the utlization of the memory on all nodes is uniform and no node's

memory is overutilized due to the prefetching mechanism.

Keeping the above issues in mind, in this thesis, we present a scheduling and data prefetching framework on Hadoop that leverages the data access patterns and memory underutilization. Our framework has been developed and implemented as a full integration into the Hadoop 2.8 ecosystem. We evaluate our framework in two modes-pseduo-distributed mode and fully distributed mode with 5 nodes on the standard WordCount benchmark. We present the results showing that our framework causes no interference with the existing Hadoop ecosystem. Our experiments show that the framework achieves improved job completion times, higher memory utilization, higher locality placement of tasks and also better overall system performance.

ACKNOWLEDGMENTS

At this place I would like to thank many people who helped me to complete this thesis successfully. First of all, Prof. Ying Lu, who supervised me throughout this research and introduced me to the challenging Job Scheduling and Data Access Patterns and Memory Locality Problem in Distributed Systems research area. I am very grateful to her for the immense amount of time that she dedicated for supervision, for her patience, continual support, motivation and knowledge leading me into the appropriate methods and techniques needed in this work. Her knowledge and enormous support has been the key to all my work, and for that I am very grateful! Next, I would like to thank my committee members Dr.Lisong Xu and Dr.Hongefng Yu, thank you for devoting your valuable time to improve the thesis.

I would like to thank all my graduate student colleagues who helped me by proofreading the write up of this thesis and for their continuous support. Next, I would like to thank other professors who I have worked closely over the past couple of years for their continuous guidance and for motivating me in my graduate career when the times were seemingly tough. Finally and most importantly, I must express my very profound gratitude to my family members, and friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them, especially my parents who have supported me by showing me the right path whenever I tend to veer off. Thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, as devices from sensor networks to corporate enterprises, vehicles and embedded devices generate increasing amount of information, large-scale distributed systems, have become a major platform for processing information for back end applications, which perform sophisticated analytics to make optimization decisions for real world problems. This can be evidenced by the fact that big data analytics tools like MapReduce[1] and Apache Spark[2] have been widely adopted by both the industry and research community for performing sophisticated data processing, analytics, storage and mining. Apache Hadoop[19] is one of the most widely used open source frameworks for real time storage and processing of large data on huge clusters.

Many data intensive computing applications are characterized by the fact that they operate on time-sensitive data and involve large computational workloads while sharing resources with best-effort latency sensitive applications. The existing state-of-the-art frameworks have not yet been able to meet the requirements set by these properties to guarantee real-time performance metrics like end-end latency, throughput, and individual application deadlines.

Recent trends in the data processing frameworks have been increasingly lean-

ing towards stream processing systems like Apache Spark[2], Twitter Heron[22] and Apache Storm[23] due to their ability to process data produced continuously within a smaller time frame. However, batch processing systems like Apache Hadoop are still heavily used and there will always be a need for them due to their ability to process complex queries over large amount of static data[24]. Hadoop is fully open source, is continuously evolving and dominates other batch processing system with its widespread usage in both industry and academic research.

A key challenge in Hadoop framework is increasing the memory utilization to improve the overall performance. HDFS[20], the underlying distributed file storage system of Hadoop, stores the data distributed across all the nodes of the cluster. Distributed processing frameworks schedule jobs (or) applications across all nodes of a cluster to maximize the utilization. Underlying schedulers in these frameworks prefer scheduling individual tasks in the nodes that contain their input data to reduce the remote I/O data copy operations which would otherwise cause data transmission overhead. In other words, placing computation near the data, is deemed important because both the network I/O and data I/O could become bottlenecks.

Another challenge in Hadoop has been the efficient usage of the network bandwidth to maintain performance of the cluster as well as provide high data throughput. The network bandwidth speeds have increased so much in recent years but it can still be a bottleneck slowing down the performance[6] and can lead to excessive resource usage in the cluster. Many workloads that run on Hadoop can arrive with a requirement of high data rate so that their deadlines are met. The contention for network bandwidth becomes difficult to handle when there are such applications requiring high throughput as well as low latency[27]. Moreover, since the application has to compete with other applications in the transfer of data between nodes, network bandwidth becomes a big contention issue[28]. One option to address the network being

the bottleneck when the cluster runs workloads that have high data rate requirement is to use network equipment with sizeable buffers which can handle the network congestion issues. Obviously, this solution is not always feasible as it comes with a high price tag and requires significant rework of the existing infrastructure.

Increasing amount of evidence shows that as the number of jobs in the cluster increases, disk and network I/O bandwidth can be a huge limiting factor for achieving desired performance[11]. In practice, the cluster resources are shared between several users that run heterogeneous applications that contend for cluster's resources. This makes it very difficult for the scheduler to assign tasks to nodes that store its input data[21]. Intuitively, the scheduler will always try to place a task in a node with the data. In some cases, the scheduler might even wait for a period of time so that the node with the data might become available in a future point of time. If the scheduler does not find a node on the clutser with the data required for the task, it will then try to place the task on a node where the data is in the same rack. If the scheduler is unsuccessful again, it will have to place the task in a node far away from the data. This mechanism induces delay(s) i) while the scheduler tries to place the task near the data and ii) also when the scheduler places the task in a node without the required data. The latter delay is due to remote data access which requires the copy of the required data for the task from a remote node where the data is physically located. This is why achieving the desired data input locality for an application is of paramount importance so as to address this latency as much as possible[4].

One mechanism that could help in addressing the aforementioned delays and bottlenecks is the data prefetching mechanism, which is a big part of my work in this thesis. Data prefetching is a data access technique that retrieves the required data for future tasks into the nodes.

As the prices for memory have started falling, clusters are being equipped with

increasingly more memory, meaning that the memory is no longer the main bottleneck in the clusters. One recent trend has been to overprovision the clusters with spare memory to reduce the probability of creating hotspots and to allow for any unpredictable memory demands by the workloads [9] . However, [10][9] have found that this memory overprovisioning leads to its high underutilization. Another study done on Facebook cluster data shows that the median memory utilization is around only 10%, leaving much room that can be leveraged for storing input data for future tasks[12]. Many studies show that inefficient memory usage is one of the major reasons for degraded performance of the cluster. Since the nodes in clusters are usually equipped with large amounts of memory, which is often underutilized, prefetching the required data for future tasks into the memory has a potential to increase the performance of the applications by reducing the effects of bottlenecks due to data I/O and Network I/O.

Although several scheduling, prefetching algorithms to hide the access delay have been proposed to improve data locality in Hadoop, there has not been much research targeting both data locality, data access patterns and real-time scheduling issues together. Some research efforts have focused on inter-block and intra-block prefetching schemes in an earlier version of MapReduce framework to improve data locality for map tasks[13]. For example, Tao Gu et al. proposed a prefetching framework for MapReduce in heterogeneous environments[17]. However, these studies do not consider the data access patterns across the cluster while prefetching and caching the required data for tasks.

Considering the data access patterns is crucial because the computation might access some portion of the data in the cluster only once while the rest could be accessed multiple times. Blindly retaining data in the memory might eventually lead to ineffcient utilization of the memory. Work done in [14] shows that 2.5% of the

files are accessed more than 10 times, 1.5% of files are accessed more than 3 times concurrently while 90% of the files are not accessed by more than one task at a time. It is also known that data popularity also changes over time and can be leveraged to predict the future access patterns. This information can be used to cache popular data blocks in the memory, to be used by the later tasks or applications. Examples of data that is accessed more frequently during computation are hash tables, data structures and objects etc. So retaining the data blocks that are accessed more could lead to increase in performance of the cluster and help the jobs and applications maintain their real-time constraints. This is due to the fact that it avoids the data and network I/O delay as there is potentially no remote data access in this case. The Namenode [30], the centralized server of the HDFS, due to its awareness of the data block accesses from the blockreports from the nodes, handles the responsibilty of marking some blocks popular across the cluster, not just a node. This makes the maintenace of metadata of popular blocks a part of the already existing bookkeeping system about the blocks in the cluster's nodes. Keeping in mind that the memory on the cluster nodes could get overutilized due to prefetching mechanism and the retention of popular blocks, we need an efficient data block replacement mechanism for the data in the node's memory such that the data not accessed frequently should be a prime candidate for eviction.

Ananthanarayanam, Ganesh et al.[14] and Abad, Cristina L. et al.[15] developed data replication algorithms that exploit the data access patterns to help the underlying scheduler to make scheduling decisions with data locality decisions. Another work by Ibrahim, Shadi, et al. [16] proposed a sheduling algorithm with the main focus to increase the data locality by reducing the amount of unbalanced execution of tasks across nodes and by predicting the next appropriate task to be placed. Even though these studies focus on load balancing and distribution of data across the cluster, we

believe exploiting these properties along with prefetching and memory locality will have more improvements in performance, especially considering the aforementioned underutilization of memory in the clusters.

In multi−core and multiprocessor environments, it has been known for the past couple of decades that having a centralized scheduler that migrates the threads across different workers leads to less frequent migration compared to a case where the workers themselves take responsibilty of actively sharing the workload [25], . This mechanism has been termed as "work stealing" and has been thoroughly studied in several studies and has been applied to "steal" resources [26], workloads[25] etc. These studies repeatedly found that the stealing mechanism leads to better load balancing and higher resource utilization.

Due to the presence of centralized scheduler in the form of Resource Manager in the Hadoop YARN system[18], the same logic could be easily extended for balancing the memory loads across the nodes in the cluster such that the utlization of the memory on all the nodes is uniform and no node's memory is overutilized due to the prefetching mechanism. Since this thesis proposes a prefetching mechanism that is data access pattern aware, the load balancing mechanism becomes crucial so that popular blocks in a node are still retained in the cluster by migration if possible, instead of evicting them when the node's memory is close to being fully utilized. Even after migration to a different node, this mechanism still retains the usefulness offered from retaining popular blocks in the memory because the blocks were deemed popular across the cluster, not just within the original node.

We propose a framework for locality aware real-time scheduling system on Hadoop that focuses on increasing memory utilization and achieving improved job completion time and system performance. The main contributions of our framework are :

1. Leverage data access patterns across the cluster and use the metrics to make caching decsions and improve the data locality
2. Make the prefetching, caching and load balancing mechanisms centrally coordinated so that the individual nodes are not "actively" participating
3. Leverage the memory locality awareness for better scheduling decisions by the Hadoop YARN system
4. Make the caching mechanism globally managed and accessible throughout the cluster
5. Replacement schemes in the cache that respect the popularity history of the blocks
6. Centrally coordinated load balance mechanism for cached data to maintain uniform distribution acorss the cluster
7. Since applications run in waves of tasks[10], we try to improve the data locality for entire waves of tasks to speed up their execution
8. We evaluate our framework on a real cluster

Rest of the thesis is structured as follows. Chapter 2 presents the background and related work, Chapter 3 presents the design and implementation, Chapter 4 presents the theoretical analysis, Chapter 5 presents the experimental methodology and Chapter 6 presents our results.

Chapter 2

Background

Hadoop (Highly Archived Distributed Object Oriented Programming) is a distributed processing framework that has been designed to process huge amounts of data spread across, potentially, hundreds to thousands of nodes in a cluster environment. Hadoop is released as an open source project in Java technology forming a part of the Apache Software Foundation umbrella, which is a non-profit open software foundation that has maintained a very important role in the distrubuted processing field [29]. Hadoop handles and allows processing of multitude of data types like audio, video, text, records, queries and even unstructured data. An important property of Hadoop is that the data is distributed across hundreds or thousands of nodes in the cluster while concurrently running computation in close proximity to their required data.

2.1 Architecture of Hadoop

Hadoop has three main components or layers as we call them in this thesis. i) Hadoop Distributed File System (HDFS), which is a distributed file system designed keeping in mind scalability, data availability ii)Hadoop YARN, a resource management and

scheduling component and, iii) Hadoop MapReduce a parallel programming paradigm for running applications to process large amounts of data in the cluster.

2.2 MapReduce

Hadoop MapReduce is data-driven parallel processing paradigm of Hadoop that runs on top of Hadoop YARN. MapReduce makes the parallelization of processing transparent to the application programmers and helps them focus on writing the data processing applications. MapReduce layer is where the programmer writes their application and where the application (or Job) is "run".

The core of MapReduce is splitting up the computation i.e, the application/Job into two phases, each containing different types of tasks. These phases are called i)Map, ii)Shuffle, iii)Reduce phases. Across these phases, there are two types of tasks in MapReduce: map tasks and reduce tasks. At first, the original input data is partitioned into several splits, with each split the size of several blocks. The map tasks in map phase consume one split each from the original input data required by the application parallely and processes the data as intended by the application. After processing the data, map tasks produce "intermediate" results, which contain a collection of key/value pairs. This intermediate data produced by map tasks is sorted according to their key values. The sorted collection of key/value pairs undergo shuffling, where the data is sent to appropriate reduce tasks as inputs. The reduce tasks then sort the received key/pairs from several different map tasks according to their key values, once this is done the final output data is written to the underlying distributed file system, HDFS. This process is visulaized in Figure 2.1.

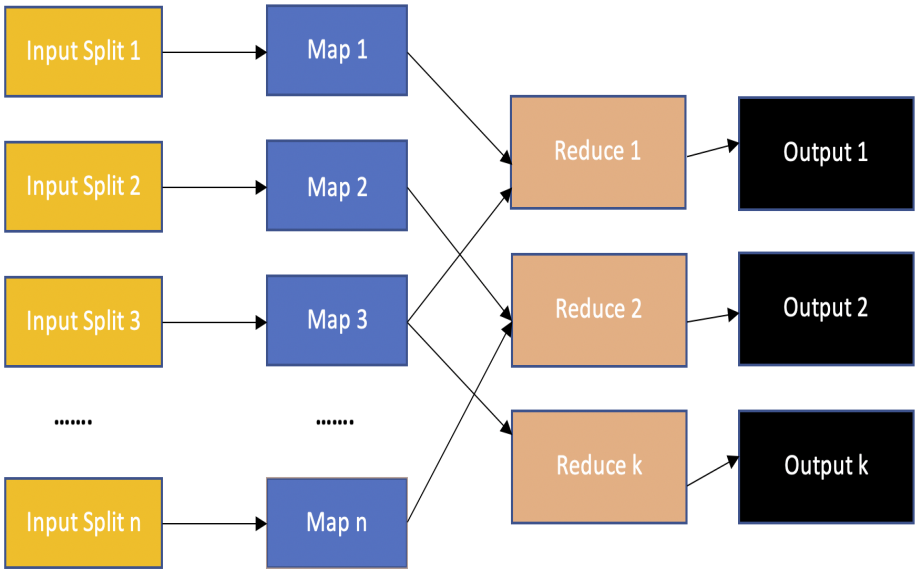

Figure 2.1: Breakdown of a MapReduce Job

The main component of MapReduce is a per-application Application Master (AM) that interacts with the YARN component for resource negotiation and runs the tasks.

2.3 YARN

Hadoop YARN (Yet Another Resource Negotiator) forms the resource management and job scheduling layer of the Hadoop architecture. YARN was introduced in Hadoop version 2.0 as a way to take off the resource managament and scheduling duties from the MapReduce into a separate component. This was done to improve the scalability of the Hadoop framework without over-burdening the MapReduce component with resource management and scheduling duties. The main changes that YARN brought into Hadoop environement are that it i) now delegates the scheduling responsibilities to a per-application component in the form of MapReduce's Application Master (AM), ii) provides dynamic allocation of cluster resources to the applications running in the cluster. This is a move away from the static allocation of resources to the ap-

plications that was implemented during Hadoop version 1. Dynamic allocation brings better cluster utilization due to its intrinsic ability to adjust the resources allocated to applications on-the-go.

The main components of YARN are the per-cluster Resource Manager (RM) and the per-node Node Manager (NM). Resource Manager runs on a single dedicated node in the cluster and takes the responsibility of management and allocation of cluster resources among the applications running in the cluster.

Resource Manager (RM) runs several components, the most important ones among them are the YARN Scheduler, Application Master Launcher and the Application Master Service (also known as Application Manager Service). YARN Scheduler is an abstract service that receives heartbeats from the nodes and also allocates requested resources such as memory, CPU, disk, network etc., to the applications running in the cluster. The resource allocation provided by the scheduler is constrained by the current cluster metrics like available capacities, waiting queues etc. YARN supports several types of schedulers like FIFO (first-in-first-out), capacity and delay schedulers. The scheduler allocates resources to the applications in the form of containers. A container is a collection of resources like CPU, memory, disk, network etc., that is required to run the application. Application Master Launcher is a thread running in RM that receives requests for new applications and then launches a corresponding application master (part of MapReduce component). Application Master Service is a thread that receives RPCs (remote procedure calls) from the application's application master (AM) and forwards their resource allocation requests to the YARN scheduler.

Node Manager is daemon that runs in all slave nodes of the cluster and acts as a slave to the Resource Manager, which is the master. The main duty of Node Manager (NM) is to monitor the resources on its host node by communicating with the AM(s) running on the host node and the containers allocated to the AM (and hence the

application). Figure 2.2 illustrates the resource negotations and application execution process. Note that, this flow involves both YARN and MapReduce components.

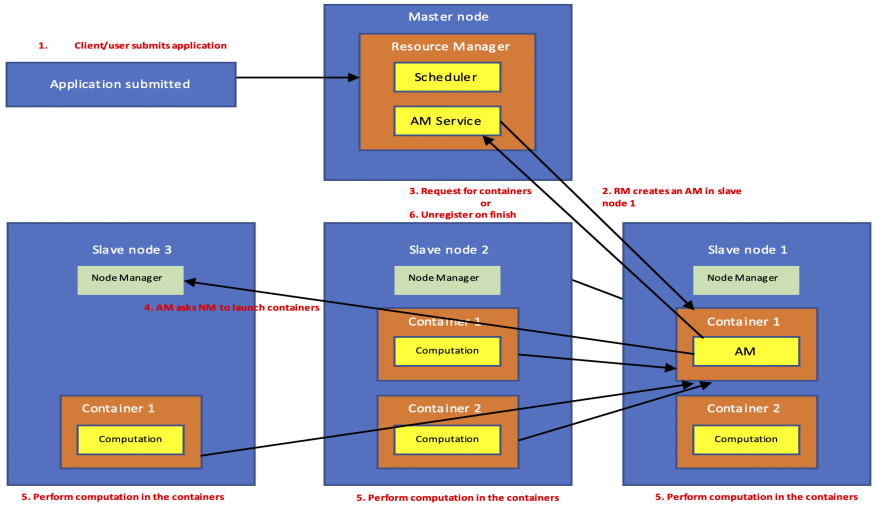

Figure 2.2: YARN Resource Negotiation and Application Execution Flow

The flow in Figure 2.2 could be explained as follows: The client or user writes an application and submits it to the RM. The AM launcher in RM launches an AM for the application in its own container in some node. The AM negotiates resources with the RM and once they are granted, AM launches the allocated containers in other slave nodes by communicating with the NM residing in those nodes. The launched containers perform the task's computation, orchestrated by the AM. NM monitors the resource usage of the containers in its node and keeps in touch with the RM via periodic heartbeats. Once computation is finished i.e., the application has finished its execution then the AM unregisters with RM and asks to deallocate the containers granted to it.

2.4 HDFS

HDFS is the main underlying open-source distributed file system of Hadoop providing scalablity, reliability, fault tolerance and availability of data. HDFS splits and distrbutes large amounts of data across thousands of machines in the cluster. To handle failure or corruption issues of data in a machine, HDFS also replicates all data in the cluster in several machines (default is 3 machines). Similar to YARN, HDFS follows a Master-Slave architecture. There are two major components in HDFS: i) NameNode and, ii)per-node DataNode.

There can be multiple NameNodes in an HDFS cluster, one per namespace. NameNode acts as a master that manages the data stored across the cluster and handles its access by the applications. NameNode manages a namespace, which is a hierarchical structure of a filesystem and directories. NameNode orchestrates the access to the files in its namespace from the applications running in the cluster. NameNode also maintains a large hashmap in its memory that contains a mapping between each DataNode under it and the HDFS blocks stored on that DataNode. A HDFS block is simply a chunk of data of certain size, e.g. 64MB. NameNode instructs DataNodes to create, replicate, delete, copy and transfer the blocks. HDFS provides the ability to maintain a secondary NameNode for the same namespace that will replace the orginal NameNode in case of a failure.

DataNodes are the slaves nodes that report to exactly one NameNode via BlockReports (Similar to a heartbeat from Node Manager to RM in case of YARN). In each report, they provide the list of blocks and their metadata to the NameNode. The NameNode updates the metadata of the blocks using information provided from all DataNodes under it. DataNodes also receive commands from the NameNode to manipulate the blocks being hosted in them. Figure 2.3 illustrates the architecture

of HDFS.

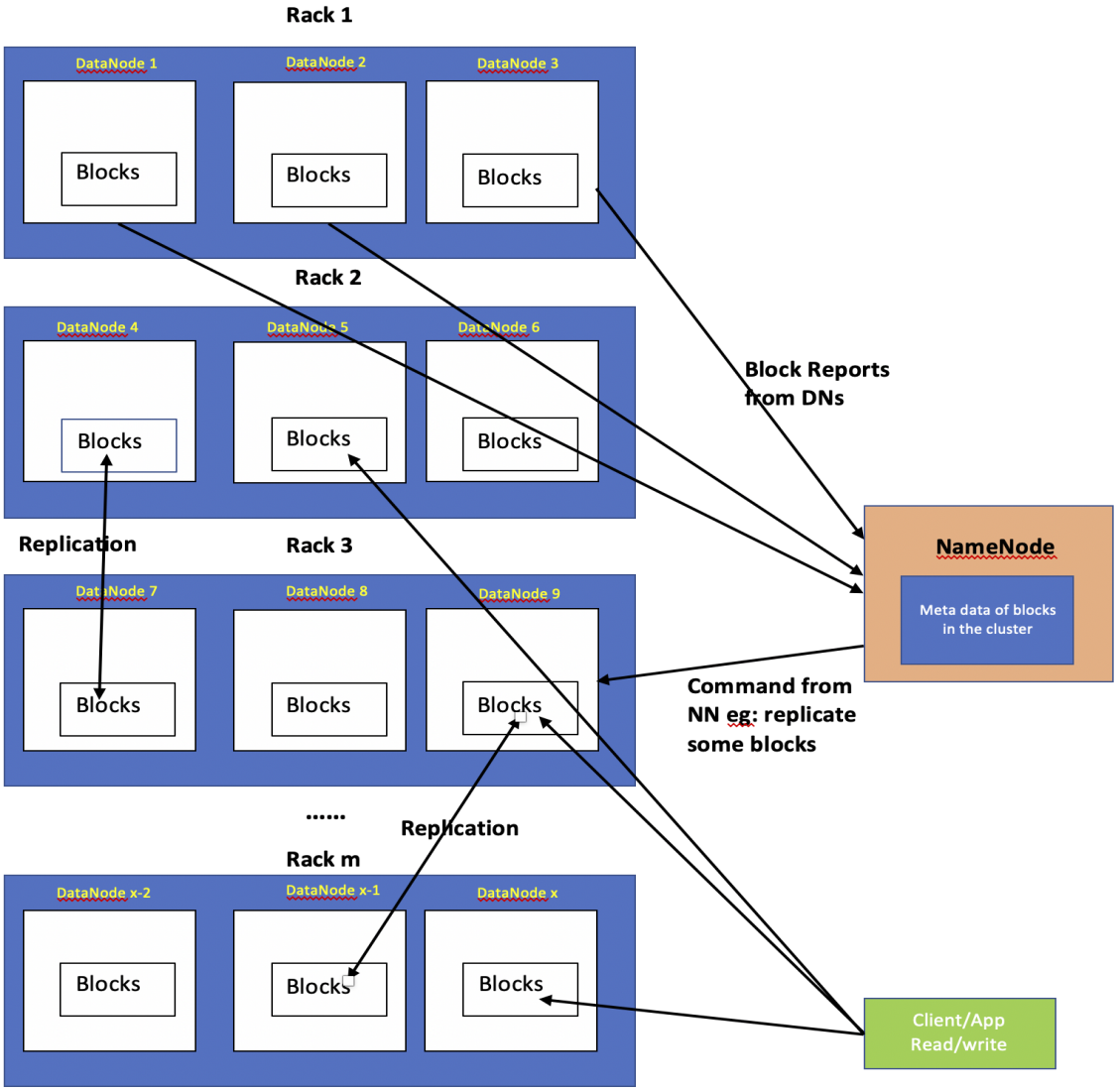

Figure 2.3: HDFS Architecture

Figure 2.3 illustrates an HDFS cluster with one NameNode, x Datanodes and m racks. Note that a rack in HDFS is a set of nodes in the cluster that are connected to the same network switch.

Chapter 3

Literature Review

For the literature review, we examine the roles of locality and memory utilization on the cluster performance and on the individual application's performance. We then consider some of the research work related to the methods aimed at improving data locality and memory utilization for the jobs in the cluster along with the methods aimed at load balancing and scheduling that form the part of such methods. Finally, we discuss the drawbacks of these research efforts.

It has been established in existing work that launching a task on a node that does not contain input data i.e., launching a remote task is inefficient because that node has to first fetch the task's required input data from the origin, i.e., from the node conatining that data. This adds an access delay that will increase the task completion time and therefore affects the system performance. In fact, data locality problem is one of the most studied issues in the field of distributed processing systems. As the number of applications running in the cluster increases, disk and network I/O bandwidth become bottleneck for achieving the desired cluster performance[11]. Additionally, the cluster runs numerous heteregenous applications from several hundreds of users and consequently all of them compete for the allocation of nodes so that they

can run their tasks. Intuitively, it has long been known that performance of the hadoop system is hugely dependent on the underlying task scheduler because it is the scheduler that decides on which node a task must be placed in the cluster. Due to the aforementioned reasons, placing the tasks on nodes that store its input data is not an easy task for the underlying scheduler [21]. To compensate for this, the native hadoop ecosystem uses delay scheduling mechanism which tries to improve data locality by first requesting to lauch tasks on nodes with input data and then wait for a certain amount of time hoping that the nodes would become available so that the task(s) could be launched on them [4]. Another well-known scheduler Quincy [35] proposed by Isard, M. et al. tries to address the data localilty issue by modelling it into a classic min-cost flow problem in a directed graph. There are several such schedulers that were studied which we cannot list due to space constraints. One major issue with these approaches is that if the scheduler does not find a task with local data on that node[31], a node maybe skipped by the scheduler for all the applications in the scheduler's queue that are waiting for their tasks to be scheduled. This issue will have an affect on the scheduler's performance and hence the application's execution time, the system performance over all and may also lead to underutlization of the cluster's resources. Another option provided by native hadoop ecosystem is FIFO scheduling system which schedules a task which is picked from a head-of-line application residing in a first-in-first-out queue of applications and launched on any node with data closest to that node. Although this mechanism prioritizes nodes with data on them, it does not make any effort in improving the data locality and hence it doesn't make any promise on increasing the throughput and guarantees almost no locality.

An early work by He, C., et al. [38] in 2011 proposed a scheduling algorithm called Matchmaking to improve a task's locality by prioritizing launching tasks on nodes with data on them while also reducing the wait time induced by the delayed schedul-

ing required for locality mechanism by setting the waiting threshold to be exactly one heartbeat interval. Ibrahim, S., et al. [16] proposed Maestro, a replica-aware scheduling algorithm for mapreduce system that aims at improving the locality of map tasks by keeping track of data block's locations along with its replicas' locations as well as the number of blocks on each node in the cluster. Maestro uses this information to schedule map tasks on nodes with the required data that causes the least amount of impact on other map tasks' that are running on nodes which also host their data.

An important aspect of our work is effectively utilizing the data access patterns for improving data locality. There have been some studies done that have focused on looking at the data access patterns during the run time of the cluster and using that information to improve data locality for the future tasks. Data access patterns provide key information about which portions of the data distributed across the nodes in the cluster is only accessed once and which portions are accessed multiple times over a period of time. So retaining the data blocks that are accessed more could lead to increase in performance of the cluster and help the jobs and applications maintain their real-time constraints. This is due to the fact that it avoids the data and network I/O delay as there is potentially no remote data access in this case. In their study[14], Ananthanarayanan, Ganesh, et al. noted that 2.5% of the files are accessed more than 10 times, 1.5% of files are accessed more than 3 times concurrently while 90% of the files are not accessed by more than one task at a time. In the same study, they proposed Scarlett algorithm which leverages the popularity of the data blocks in HDFS by replicating the popular data blocks to address the bottleneck issues caused by hotspots. Another work by Palanisamy et al. [34] proposed Purlieus, a mapreduce resource allocation algorithm that aims at improving data locality among tasks while reducing the network overhead required for the remote data transfer. Purliues places the data on nodes that will run the task that require that data or at least place the

data close to the node that will run the task. Abad et al. [33] presented a distributed adaptive data replication algorithm which determines the number of replicas a file in HDFS needs to be replicated and the locations that it needs to be replicated at based on the popularity and access patterns. Although these research efforts leverage the data access patterns, their replication schemes are not run-time and the data needs to be replicated ahead of the application's scheduling. Additionally, these studies do not make an effort to make the popular data accessible to other nodes across the cluster and they do not consider the fact that popularity of data is ever changing. It is important to note that data access patterns for any piece of data is constantly changing as newer applications are being added to the cluster and as the existign applications are making progress continuously. Since it is known that data popularity also changes over time and can be leveraged to predict the future access patterns. This information can be used to cache popular data blocks in the memory, to be used by the later tasks or applications. Examples of data that is accessed more frequently during computation are hash tables, data structures and objects etc.

Choi D, et al. [37] proposed a task scheduling algorithm that categorizes the tasks based on the the location of data blocks in their input splits and then sequentially launching tasks on nodes according to their priority that was calculated based on data locality of their input splits. The main contribution of this work is to reduce the performance degradation caused by copy of the data blocks in the task's input split that are spread across several nodes. Zhang, X. et al. [39] proposed a next-k-node scheduling mechanism that calculates the probability for each task to be launched on a node such that the tasks with input data on the next k nodes have lower probabilties than other tasks. The scheduler then launches a task on a node that has its input data on that node. If it does not find such tasks for that node, it launches a task with highest probability on that node. In 2016, Wang et al. [31] proposed a task

scheduling algorithm that aims at improving data locality by considering a tade off between data locality for a task and the consequent load balancing. Although the study does a good job of considering the load balancing aspect involved to achieve the desired data locality, the limitation of the study is that it only addresses the optimization of load balancing from a network perspective but it does not address the issue of improving data locality itself by considering the data access patterns.

One mechanism that has been studied for quite some time is the idea of data prefetching, in which the required data is fetched to the node in advance where the task is to be executed. This idea is been aimed at improving data locality as well as reducing delays in the task's execution caused by data copy over the network and thereby reducing the application's execution time while improving performance. Seo, S. et al. [13] proposed HPMR, a prefetching and preshuffling mechanism for the earlier version of mapreduce system (when YARN did not exist yet). The prefetching mechanism prefetches an input split or data blocks for both map and reduce tasks. The preshuffling mechanism predicts where the reduce task might be placed after the map task finishes and based on this prediction it tries to increase the amount of data that is shuffled over the network in advance. Another work done by Tao Gu et al. [17] proposed a prefetching framework that models the problem as a form of producer-consumer problem, where the task running on a node without its input data on that node triggers a prefetch mechanism that prefetches data in a buffer from one end while the task processes the data from the other end of the same buffer. The limits of these two studies are that they do not aim at reducing the amount of tasks that are remote i.e., do not have required data on their nodes as they do not exploit the data access patterns. Additionally, they do not consider the data copy delay for prefetching the first chunks of input data.

Some work started leveraging the fact that memory is underutilized in the clusters

and thus can be used for storing input data for future tasks. It must be noted that nodes in the clusters are being equipped with large amounts of memories as the memory price continues to fall. The work done by Ananthanarayanan et al. [10] on analysis of Facebook and Bing datacenters showed that on average 79% of the application/job's duration is spent on data copy operations. The study also found that the median and 95th percentile utilization of the cluster memory falls at 10% and 42% respectively. Sun, M. et al. [12] proposed a prefetching based scheduler for mapreduce system called HPSO which aimed at improving memory locality in the cluster. HPSO relied on the fact that prefetching accuracy improves the performance of the system. HPSO predicts which tasks will be assigned to which nodes and then uses this information to prefetch the data required by the task into the node's memory thereby effectively overlapping the data transfer delay with the computation and thus hiding the delay from data transfer. However, a limitation of this work is that once the input data is prefetched and then cached into a node, there is no discussion on what happens to this data after the task is done processing on that data. This drawback also exists for the studies [13] and [17] that were mentioned in the previous paragraph. Also, HPSO and the aforementioned studies do not make any effort to consider the memory utilization imbalances caused by prefetching mechanism and neither do they consider any data retention or load balancing schemes for the prefetched and already processed data residing in the node memory of the cluster.

There were some studies that introduced a centrally coordinated shared memory system that exploits the cluster memory's underutilization. One such work was done by Hwang, J. et al. [9], where they presented that most clusters overprovision their memory so that they can handle unpredictable bursty workloads that might occur in the future. Their study showed that 50% of the nodes in their cluster consisting of 50 nodes that have at least 30% of memory being underutilized as a consequence

of memory overprovisioning. Several other studies such as Ananthanarayan et al. [10], which is noted in this section, showed that overprovisioning the cluster memory leads to its high underutilization. In order to exploit this memory underutilization, Hwang, J. et al. proposed the Mortar framework, which runs a hypervisor process that repurposes the underutilized sections of memory in the cluster for storing task related data that is generated during run time. This run time data is essentially a prefetched data that can be an input for either a single task running on a node or for the entire application distributed across several nodes managed by a separate protocol. The nodes can freely add any spare memory to the shared memory managed by the hypervisor process which coordinates its usage for the entire cluster for storing prefetched data. The hypervisor can evict data from the shared memory if a node gets constrained in its memory usage. When it comes to exploiting the underutilization of memory in the cluster, the approach taken by this work is related to the approach taken in our thesis where we introduce a centrally coordinated process that manages the free memory as well, albeit our central process has added several functionalities with two notable functionalities being that our process leverages data access patterns to decide which data to cache and it actively manages which node has how much free memory instead of letting each individual nodes add the free memory.

There are a couple of recent works done in 2018 that are more closely aligned to the load balancing aspect employed in our work in this thesis that aims at balancing and maintaining a uniform utilization of memory across the nodes in the cluster. Load balancing aspect for maintaining uniform utilization of memory across the nodes in the entire cluster is important so as to not let any of the nodes go overutilized thereby affecting the cluster performance and to avoid hotspots.

Li, C. et al. [36] proposed a two mechanisms, one aimed at improving data locality by data migration or prefetching and the second aimed at determinging popular files

or "hot" files as the authors call them and sharing those hot files between multiple data centers of a geo-distributed cloud environments periodically. In [33] Li, C. et al. presented a scheduling aware data prefetching mechanism for hadoop based hybrid cloud systems which fetches the task's required input data to the local node before the task is executed. The work also presents a file synchronization mechanism which syncs only popular files between subcloud environments. The major drawback of this work is that they only consider the data access patterns for syncing files between different subcloud environments. The nodes within the subcloud i.e., nodes within a single (large) cluster system do not have direct access to the popular data and hence the effect of their file synchronization mechanism in improving the performance of the nodes within one single large cluster system is lacking. Additionally, although this work considers the data access patterns of the files, they do not consider the data access patterns of data at the block level. Finally, these two studies do not address the issue of what should be done to the prefetched data. This aspect is important to avoid imbalances in memory utilization. We need an efficient centrally coordinated data block eviction/replacement mechanism for the data in a node's memory such that the data not accessed frequently should be a prime candidate for eviction thereby retaining more frequently accessed data. Our work in this thesis includes such a data eviction scheme.

As you can see there are several scheduling and prefetching algorithms focused on improving data locality and hence the system performance, there has not been much research that addresses data locality, data access patterns and real-time scheduling and load balancing issues together. To the best of our knowledge, our work is the first one that considers addressing these issues in a single framework and we propose that exploiting these properties together offers more improvements in performance.

Chapter 4

Design

4.1 Introduction

When the Application Master (AM) is initialized in one of the nodes by the Resource Manager(RM), it has to request the containers in the nodes while considering which nodes offer best locality for the application. For this, AM has to know which nodes have the required blocks stored in their local memory or disks. Maintaining this information of the local memory is not optimal and comes with a heavy cost because the local memory tends to change frequently. In the approach taken in this thesis, local memory in each node is divided into two portions: local cache and global cache. The most popular and young (which are newly created blocks) blocks are stored in the global cache while the local cache stores the blocks that are needed to run the tasks. Due to the nature of the blocks stored in the global cache, its content does not change as frequently as local cache's and the information about the blocks in the global cache is maintained at the NameNode (NN) of the HDFS layer. The locality information of the blocks in the global cache is made available to application masters. This information can be used by an application master to make better locality decisions

while requesting the containers.

For tracking the popularity of blocks across the cluster, we define several popularity metrics for all blocks in the cluster. The variable NP_i, called node popularity, keeps track of the total number of accesses to $block_i$ in a node since its creation in the local memory. The variable AP_i, called application popularity, keeps track of the number of accesses made to the $block_i$ by this application. The AP_i variable is uniquely maintained for a $block_i$ across all nodes that this application master has tasks running on them. For this reason, this variable is updated only once the application finishes. The variable GP_i, called global popularity, keeps track of the number of accesses made to $block_i$ across the cluster. Additionally, a reference bit R_i is maintained for each block and is set whenever the block is accessed. This reference bit is used to evict least recently accessed blocks from memory with the modified clock algorithm.

AM knows that it needs to request $\boldsymbol{w}$ number of containers for its application. AM first requests locations of the needed blocks in the cluster from the NameNode. AM then receives a list of nodes with needed blocks in their global cache, disks and racks. After the AM knows about the blocks and their locations in the global cache from global cache image, AM then sends the list of nodes that it wants the containers to be assigned on to the scheduler that runs at the RM. But first AM needs to make decision about which nodes to request the containers on.

4.2 Containers Request Algorithm

The main goal of AM at this stage is to offer the highest locality possible so it needs to request containers on the nodes that provide the highest locality for the application. AM requests the containers on a list of nodes with data on them in their global caches

Algorithm 1

```
1: procedure GET_LOCATIONS
2:     Input: List of blocks B that are needed by the Application
3:     Output: Locations, which is the mapping between list of blocks to node global
   cache/node disk
4:     for each block B_i in B do
5:         Add locations of B_i in Global_Cache_Image to Locations
6:         Add locations of B_i in HDFS to Locations
7:         Initialise the application popularity AP_i = 0
       return Locations
```

or disks. The nodes with the highest number of needed blocks in their memory have the highest priority. Ties are broken with the highest number of needed blocks in the rack's memories, the node's disk and then the rack's disks. First, nodes in the cluster are ranked according to their locality information stored in their respective node rank NR_j for node N_j.

While considering the nodes to request containers on, first the nodes in the cluster are ranked according to their locality information stored in their respective node rank NR_j for node N_j. Node rank NR_j of node N_j is represented as a vector of length 4 : **(a,b,c,d)**. **a** is the number of blocks of the application stored in node N_j's memory. Note that the local cache content is not known to the application master until it launches a container so **a** is the number of blocks in the node's global cache only. **b** is the number of blocks in global cache memory of nodes on the same rack as N_j. **c** is the number of blocks in the disk of N_j. **d** is the number of blocks in the disk of nodes on the same rack as N_j. Note that a block is only considered once in calculating a,b,c,d and its replicas are not counted.

Assuming that we have two nodes N_1 and N_2, N_1 has a higher node rank than N_2 i.e., $(a_1, b_1, c_1, d_1) \geq (a_2, b_2, c_2, d_2)$ if $a_1 > a_2$ or ($a_1 = a_2$ and $b_1 > b_2$) or ($a_1 = a_2$ and $b_1 = b_2$ and $c_1 > c_2$) or ($a_1 = a_2$ and $b_1 = b_2$ and $c_1 = c_2$ and $d_1 > d_2$) or ($a_1 = a_2$ and $b_1 = b_2$ and $c_1 = c_2$ and $d_1 = d_2$). AM must decide (N_1, X_1),

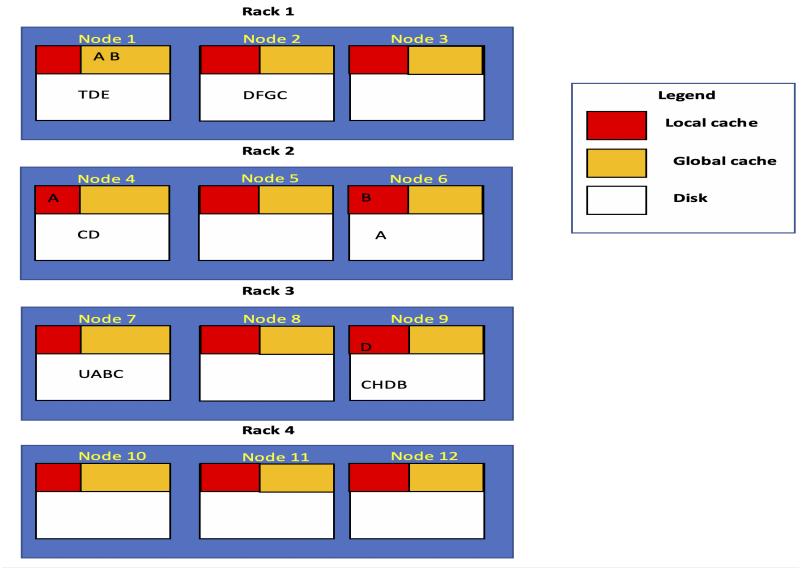

Figure 4.1: An Example Scenario of Task Assignment Based on Memory Locality

$(N_2, X_2).....(N_j, X_j).....(N_k, X_k)$ where (N_j, X_j) denotes requesting X_j number of containers on node N_j.

Consider the cluster shown in Figure 1, the application has five tasks T_1, T_2, T_3, T_4, T_5, each needing the blocks A, B, C, D, E respectively. Suppose for this job $\boldsymbol{w}$ is 3 so the application master requests three containers. The node ranks of the cluster are $NR_1 = (2, 0, 2, 1)$, $NR_2 = (0, 2, 2, 1)$, $NR_3 = (0, 2, 03)$, $NR_4 = (1, 1, 2, 0)$, $NR_5 = (0, 2, 0, 2)$, $NR_6 = (1, 1, 0, 2)$, $NR_7 = (0, 1, 3, 1)$, $NR_8 = (0, 1, 0, 3)$, $NR_9 = (1, 0, 3, 1)$, $NR_{10} = (0, 0, 0, 0)$, $NR_{11} = (0, 0, 0, 0)$, $NR_{12} = (0, 0, 0, 0)$. AM adds N_1 to the request as it has the highest number of blocks in local memory and updates NR_1 to be $(1, 0, 2, 1)$. After that N_4 becomes the node with the highest node rank. Similarly N_6 is added to the request as the host of the final container.

After adding a container to the request, the node rank is updated by reducing the number of blocks in the memory by 1 if it is non zero. Otherwise we decrease the number of blocks in rack global cache, the number of blocks in local disk or the number of blocks in the rack's disks by 1 whichever is the first to be found non zero.

Algorithm 2 Containers Request

1: **procedure** REQUEST_CONTAINERS
2: Input: none
3: Output: List of containers to request
4: Calculate NR_j for all N_i in N
5: **while** request_size *less than* **w do**
6: Select N_j with the maximum node rank (a,b,c,d) i.e. the node N_j where $\forall$ k $k \neq j$, we have $NR_j \geq NR_k$
7: **request.add**$(N_j, 1)$
8: UpdateNR(N_j)
9:
10: **procedure** UPDATENR
11: Input: Current NR_j (a_j, b_j, c_j, d_j)
12: Output: Updated NR_j (a_j, b_j, c_j, d_j)
13: **if then**$a_j \neq 0$
14: a_j- -
15: **else** If $b_j \neq 0$
16: b_j- -
17: **else** If $c_j \neq 0$
18: c_j- -
19: **else**
20: d_j- -
21: return NR_j

4.3 Container Assignment Algorithm

After AM requests the containers from the RM, the scheduler at RM has to decide which application needs to be assigned which conatiners on the nodes of the cluster.

Once the scheduler at RM receives the AM's container request, it starts to make scheduling decisions to decide which nodes to allocate the containers on from the list of nodes received from AM in its request. The scheduler maintains i) a list of applications that sent container requests to it and are now waiting for containers and, ii) a list of nodes in the cluster that are available for placing containers on them.

Whenever a node becomes avaliable to be allocated a container(s) on (via node update message sent to the RM by the node), the scheduler needs to make the decision

on which application(s) to grant the container(s) on that node. Our main idea is to launch containers on the node for an application that has the highest probability of its tasks being processed non locally if not being chosen. We introduce a value called Block Weight (BW) to represent the likelihood that the block is processed locally.

The following are the steps involved in calculating block weight for a block. First, the related blocks of node j are divided into four categories : α_j are the application's blocks in the local memory, β_j are the application's blocks in the global cache of nodes located on the same rack, γ_j are the application's blocks in the local disk of the node and δ_j are the application's blocks in the local disk of the nodes located on the same rack.

An individual gain of processing a block on a node is either 10, 8, 4 or 2 depending on whether the block falls into α_j, β_j, γ_j or δ_j category respectively. Let τ_{ij} be the individual gain of processing block i in node j, then the relative gain of processing $block_i$ on node j is the ratio of the gain τ_{ij} to the summation of individual gains of all application's blocks on node j. The block weight BW_i can be calculated as the summation of relative gains of processing block i on other nodes with containers.

Block weight for block i is $BW_i = \sum_{N_k \leftarrow \phi - N_j} \tau_{ik}/(10 * size(\alpha_k) + 8 * size(\beta_k) + 4 * size(\gamma_k) + 2 * size(\delta_k)) - 8 * size((\alpha_k \cap \beta_k)) - 4 * size((\alpha_k \cup \beta_k) \cap \gamma_k) - 2 * size((\alpha_k \cup \beta_k \cup \gamma_k) \cap \delta_k)$, where size() gives the size of the set, ϕ represents all the nodes with containers for the application.

In the above expression, the numerator τ_{ik} represents the gain τ for processing $block_i$ on node k. If the block is in the local memory τ_{ik} is 10, if the block is in rack memory then τ_{ik} is 8, if the block is in node k's disk then τ_{ik} is 4 and if the block is in rack disk then τ_{ik} is 2. The denominator represents summation of gain in processing some "local" blocks on node k. The lower the ratio (also called relative gain), the less likely that block i will be picked to run on node k locally. Thus, a lower block weight

indicates that the block has a higher probability of being processed non locally if not being chosen to run on node j.

The block weight is calculated for all blocks in the current node and the nodes in the same rack. Once block weights for all blocks are calculated for the application, the average block weight for that applicaton is calculated. Similarly the scheduler proceeds to calculate the average Block Weights (BW) for each application on that node and puts the application-BW pair in a list. Once the scheduler calculates the average BWs of all the applications waiting for container(s) on this node, it picks the application with the lowest BW on that node and assigns the maximum possible containers for that application on the node. This ensures that we assign containers to the application on this node with its tasks having the highest probability of being processed non locally if not being granted containers on this node. Note that maximum possible containers on a node is the minimum of number of requested containers on the node by the application and the maximum allowed number of containers on the node. The latter depends on the cluster configuration and settings.

The following algorithm describes the container assignment process discussed above.

Algorithm 3 Container Assignment

1: **procedure** CONTAINER_ASSIGNMENT
2: Input: List of applications $\{application_1.....application_k\}$ waiting for containers on node N_j
3: Output: Containers granted to one application from the list
4: **for** each application $applicaton_i$ in the list $\{application_1.....application_k\}$ **do**
5: Calculate $average_{BW_i}$ for $applicaton_i$
6: $Application\_BW\_List.put(applicaton_i, average_{BW_i})$
7: **while** N_j can host more containers **do**
8: Pick the application $application_l$ with lowest $average_{BW}$ from $Application\_BW\_List$
9: Launch maximum allowable containers on N_j for $application_l$

4.4 Execution and Data Block Prefetch

After the containers are granted, AM will launch tasks on the granted containers. Note that AM will be aware of the local cache contents in the nodes that it has launched containers in. This local cache information will also be included in making task dispatch decisions.

When the task begins processing the blocks during its execution phase, the block needs to be brought into local memory if it is not in the local memory already. If the number of blocks in the local memory is above the maximum threshold then some of the blocks need to be evicted. We denote M_j as the number of blocks in node N_j's local memory, M_{max} is the maximum threshold of the local memory of a node. The local cache eviction algorithm is called when a block needs to be fetched or prefetched and when $M_j = M_{max}$.

Once a task is dispatched, a prefetch instruction is also called to fetch a block into node N_j's memory.

Algorithm 4 Task Execution

1: **procedure** TASK EXECUTION
2: Input: Node N_j
3: Pick the first block needed B_i
4: **if** the block not in N_j's memory **then**
5: Call Local_Cache_Eviction(N_j) if $M_j = M_{max}$
6: Fetch B_i into N_j's local cache, M_j++
7: Process block B_i in node N_j and make reference bit $R_i = 1$
8: AP_i++
9: $NP[i, j]$ ++
10: **Prefetch**(N_j)
11: **procedure** PREFETCH
12: Input: N_j
13: Output: One block that is chosen to be prefetched into node N_j's local cache
14: U $\leftarrow$ list of blocks needed by AM that are not in local cache of nodes where AM has the container(s)
15: **if** U not null **then**
16: Select first block p from U
17: Remove p from U
18: **if** $M_j == M_{max}$ **then**
19: **Local_Cache_Eviction**(N_j)
20: Instruct N_j to fetch block p
21: M_j ++

4.5 Local Cache Eviction Strategy

Algorithm 5 Local cache eviction

```
1:  procedure LOCAL_CACHE_EVICTION
2:      NP_total = 0
3:      for i = 1 to M_j do
4:          NP_total += NP_i
5:      NP_avg = NP_total / M_j
6:      Eflag = 0
7:      for each block B_i where NP_i ≤ NP_avg do
8:          if R_i equals 0 then
9:              Evict B_i
10:             M_j- -
11:             Eflag = 1
12:             break
13:         Else
14:         R_i ← 0
15:     If Eflag = 0 go to 7
```

The above algorithm describes the local cache eviction strategy. If there is no space in the local cache for a new block then an unpopular block that has not been recently used is evicted. We categorize the unpopular blocks as the blocks whose node popularity is below average value. The average node popularity of the blocks in the node is calculated at line 4 and is denoted by NP_{avg}. Then an unpopular block is evicted by running the clock algorithm, otherwise known as second chance algorithm defined from lines 5 to 15. Note that the reference bit R_i is updated in line 7 in procedure Task_Execution in algorithm 4.

4.6 Global Cache Eviction and Load Balancing

After AM is done executing all the tasks in the application, it has to decide if the blocks processed by it should be moved to the global cache. Since the global cache

maintains young and popular blocks, AM needs to check if the blocks are both young and popular. A block is considered popular if its global popularity GP is above the global average popularity GP_{avg}. First, the global popularity of blocks should be updated by adding the number of accesses made to the blocks by the application executed by AM, this is done by a call to the procedure UpdateGlobal. Then all the blocks whose global popularity is above global average are moved to global cache. If none of them are young and popular then the blocks are just left in the local cache. The algorithm Finish describes this process and it is executed when AM has finished executing the application.

Algorithm 6 Finish

```
1:  procedure FINISH
2:      Create a list containing all blocks in the local memory of nodes with AM's
    containers and their respective application popularity AP called AP_list
3:      GP_list ← UpdateGlobal(AP_list)
4:      for each block B_i do
5:
6:          if GP_i > GP_avg && creation time of B_i < 24 hours then
7:              MoveToGlobal(B_i)
8:  procedure UPDATEGLOBAL
9:      Input: AP_list, list of blocks and their access counts by an application
10:     Output: updated global access count
11:     for each block B_i in AP_list do
12:         GP_i+ = AP_i
13:         GP_list.put(GP_i)
14:     GP_total = getGP_total()
15:     GP_avg = GP_total/N*G_max ; where N is the total number of blocks in the global
    cache of nodes across the cluster
16:     return GP
```

Note : The method $getGP_{total}()$ returns the total sum of GP values of all nodes across the cluster.

If any block has been moved into the global cache by the AM(line 7, algorithm 6) then we invoke global cache maintenance algorithm (i.e., algorithm 7). The goal of

Global_Cache_Maintenance is to evict old and unpopular blocks from global cache and to maintain global cache size across all nodes less than the threshold G_{high}.

First, the Maintenance procedure calculates and checks if the average global cache size (G_{avg}) is above the threshold (G_{high}). If G_{avg} is above G_{high} then it will be brought down by evicting old and unpopular blocks by calling Global_Cache_Eviction procedure (lines 3 and 4).

The eviction procedure Global_Cache_Eviction first removes all the old blocks which are older than 24 hours. After that, if the new G_{avg} (updated in line 25) is above the lower bound G_{low} then the least recently used unpopular blocks are evicted until G_{avg} reaches the lower bound.

We classify unpopular blocks as the blocks whose global popularity GP is below the global average GP_{avg}, this check is made in line 29 of algorithm Global_Cache_Maintenance. Then the *least recently used* blocks among these unpopular blocks are evicted by running the clock algorithm outlined in lines 29 to 36.

After evicting the old and unpopular blocks, load balancing is done in lines 5-17 in procedure Maintenance to make every node's global cache size equal to or less than threshold G_{high}. This is done by moving the blocks from the heavily loaded nodes N_Heavy to the lightly loaded nodes N_Light. The lightly loaded nodes are the nodes whose global cache data size is less than G_{high} and heavily loaded nodes are the nodes whose global cache data size is bigger than G_{high}.

Algorithm 7 Global cache maintenance

procedure MAINTENANCE
  Input: Global cache size of Node j G_j and $G_{avg} = \sum_{j=1}^{n} G_j/N$
  if $G_{avg} > G_{high}$ **then**
    Call Global_Cache_Eviction()
  for All nodes N_j **do**
    if $G_j > G_{high}$ **then**
      Add N_j to N-heavy
    Else if $G_j < G_{high}$
    Add N_j to N-Light
  for every N_j in N-heavy **do**
    for all B_i in N_j **do**
      Add B_i into M , do $G_j - -$ until G_j *equals* G_{high}
  while M not empty **do**
    remove a block from M and add it to the first node N_k in N-Light
    G_k++
    if $G_k = G_{high}$ **then**
      remove N_k from N-Light
procedure GLOBAL_CACHE_EVICTION
  Output: Old and unpopular blocks are evicted
  for every N_j in the cluster **do**
    for all B_i in N_j's global cache with *creation_time* < now - 24hrs **do**
      Move B_i into local cache
      G_j- -
  Update G_{avg}
  $H = (G_{avg} - G_{low}) * N$
  while H > 0 **do**
    for Each B_i with $GP_i < GP_{avg}$ **do**
      if R_i *equals* 0 **then**
        Move B_i into local cache of its node N_j
        G_j- -
        H- -
        break
      Else
      R_i equals 0

Chapter 5

Implementation

The main algorithms described in Chapter 4 are semi-standalone in that they could be "turned off" or "turned on" by the user although they were designed to be part of the core of Hadoop framework. The advantage of being able to turn on or off the algorithms is that the cluster performance remains unaffected. The ability to turn on or off the algorithms extends to individual slave nodes in the cluster as well. However, it is important to note that if at least one node in the cluster runs the algorithms then the RM, AM and NN must run the algorithms as well. This is because of Hadoop's intrinsic nature of providing centralized services to the slave nodes via RM, NN and AM.

Algorithm 1, which receives the locations of required blocks from the HDFS side, is implemented as a part of the MapReduce framework (application layer of Hadoop). Although the algorithms run at the MapReduce framework, they do require turning on the extra added functionality at the HDFS layer because the AM requires additional information offered by the NN from HDFS layer. The HDFS layer has been modified to support the awareness of blocks in the memory of nodes and several related metrics mentioned in Chapter 4. The algorithm 1 involves communication

between MapReduce and HDFS but this communication is implemented by the native protocol in Hadoop. The mechanisms at both ends are completely independent hence it does not affect the performance of either the application or AM, which runs at the MapReduce layer, or the components running in the HDFS layer like the NN or the Datanodes (DN).

Algorithm 2, used for requesting containers to run application's tasks, is part of the AM in MapReduce and involves communication with the RM at YARN layer using the native Hadoop protocols. Algorithm 2 requires additional functionality in YARN but other than that, the mechanisms at both ends run completely independently of each other. So, the performance of both MapReduce and YARN remain unaffected from each other.

Algorithm 3 assigns containers to the AM and it runs completely at the RM in YARN. Like algorithms 1 and 2, this algorithm relies on the existing protocols for the communication between AM and RM, and is stand alone so it does not affect other layers.

Algorithm 4 contains two parts, part 1 invloves task execution at the MapReduce layer and part 2 involves prefetching of data done at the HDFS layer. We modified the task execution process orchestrated by the AM to trigger the prefetching of required data at the HDFS layer. This setup has been developed such that the prefecthing does not induce any delay to the task execution.

Algorithms 5, 6 and 7 run completely in the HDFS layer. They are completely transparent to other layers and hence does not affect the application execution or the container assignment/resource management of the cluster nodes. These algorithms deal with eviction of blocks from node's memories, load balancing of blocks in the cluster and maintenance of popularity metrics of the blocks.

All algorithms were written in java using the API that was used for building the

core of Hadoop. The algorithms were implemented and tested in Hadoop 2.8.1 and Hadoop 2.9. The implementation is compatible with Hadoop versions 2.6 and above until the current version of Hadoop, which is Hadoop 3.0. We chose not to implement our algorithms in Hadoop 3.0 due to the existing bugs in the Hadoop 3.0 release. Moreover, our algorithms are completely compatible with Hadoop's existing caching mechanism, which lets users explicitly pin entire files into a node's memory.

Chapter 6

Evaluation

In this section, we evaluate the framework proposed in the thesis. In Chapter 6.1, we describe the experimental setup used to perform our experiments. Then we outline the methodology used to design our tests and collect the results. In Chapter 6.2, we present the results generated from our test cases, compare those results visually with that of default hadoop system and then we analyze and compare the results and perform subsequent discussions.

6.1 Experimental setup

The experiments are designed to evaluate the percentage of locality improvement for the tasks, task execution times, total job completion time, total tasks killed and performance improvement compared to the default hadoop ecosystem. Further, to establish confidence in our model, we conduct our experiments in two different set ups: pseudo-distributed mode setup and fully distributed mode setup. For each of these cases, we discuss and analyze the drawbacks of the hadoop system and how our approach compensates for these drawbacks. Additionally, we also discuss some of the

drawbacks presented by our framework and discuss how they could be improved in the future as well as how hadoop compensates for these drawbacks.

We have implemented the framework as part of the Hadoop 2.8.1 and the comparisons are made against the default version of hadoop 2.8.1. For each of the test cases, we performed our evaluations with wordcount, one of the most commonly used benchmarks for evaluating the performance of hadoop system. We conduct experiments for the wordcount benchmark on two different workloads. The first one is taken from an open source project called Project Gutenberg [40], which is a collection of over 58,000 free eBooks. From Gutenberg Project, we collected 2 sets of workloads. The first one contains 85 text files with sizes ranging from 41 KB to 671KB, which we name as "GutenbergSmall". The second set in Gutenberg, which we name "GutenberLarge", contains seven individual input files with their sizes varying from 8.7MB to 2.27GB. The second workload was taken from Blog Authorship Corpus [41], which is a collection of 681,288 blog posts organized into 19,320 files (one file for each user) gathered from blogger.com in August 2004. The workload consists of over 140 million words in total and the file sizes range from 1KB to 2.7MB. Additionally, we have created a small set of larger workloads with sizes 4.5GB, 10GB and 20GB. The workloads are described in tables 6.1 to 6.6.

Table 6.2: Configuration of GutenbergSmall workload overall

ID	**Workload**	**Total Input Size (Bytes)**
1	Wordcount	7862392

Table 6.1: Individual configurations of files in GutenbergSmall workload

ID	Workload	Input Size (Bytes)
1	Wordcount	133753
2	Wordcount	148572
3	Wordcount	273967
4	Wordcount	429807
5	Wordcount	98212
6	Wordcount	368952
7	Wordcount	110426
8	Wordcount	41155
9	Wordcount	114878
10	Wordcount	114379
11	Wordcount	360031
12	Wordcount	392387
13	Wordcount	392387
14	Wordcount	461521
15	Wordcount	166625
16	Wordcount	105427
17	Wordcount	325285
18	Wordcount	38273
19	Wordcount	124367
20	Wordcount	309320
21	Wordcount	123684
22	Wordcount	393344
23	Wordcount	116330
24	Wordcount	202188
25	Wordcount	288195
26	Wordcount	119559
27	Wordcount	266759
28	Wordcount	671233
29	Wordcount	82920
30	Wordcount	385159
31	Wordcount	97478
32	Wordcount	976429

Table 6.3: Configuration of GutenbergLarge workload

ID	Workload	Input Size
1	Wordcount	2.61 GB

Table 6.4: Configurations of files in Blog Authorship Corpus workload

ID	Workload	Input Size
1	Wordcount	401.6MB
2	Wordcount	404.5MB

All 19320 input files in the Blog-Corpus workload were combined into two separate input files to avoid creating 19320 mappers in the system

Table 6.5: Configuration of the Blog Authorship Corpus workload overall

ID	Workload	Input Size
1	Wordcount	806.2MB

Table 6.6: Configurations of the three larger workloads

ID	Workload	Input Size
4GBWorkload	Wordcount	4.23GB
10GBWorkload	Wordcount	10.3GB
20GBWorkload	Wordcount	20.06GB

6.1.1 Pseudo-distributed Setup

For the pseudo-distributed setup, we chose a computer with 6th generation quadcore intel i7 processor and 16GB memory. Pseduo-distributed mode basically hosts a virtual machine that runs all the hadoop daemons on the same machine and is the first

place where hadoop developers perform their experiments. This mode is a bridge between the standalone mode and the fully distributed mode. Standalone mode is not tested for our framework as it does not support running HDFS and YARN, which are required to test our framework. Pseudo-distributed mode simulates a real cluster on a single machine and provides a veritable test enironment for our experiments. For the pseudo-distributed setup, we perform experiments by running each job separately as well as running multiple jobs concurrently. Tables 6.1, 6.2 and 6.3 show the configurations of the jobs in the experiments for each of the two sets of aforementioned workloads. Both tables list all the jobs that were run separately and together. We configured the system to run at most 4 map tasks and 4 reduce tasks in parallel for each job by setting the configuration parameters *mapreduce.job.running.map.limit* and *mapreduce.job.running.reduce.limit* in hdfs-site.xml. For the pseudo-distributed mode we performed experiments with two different replication factors for the HDFS blocks, which are 1 and 3 (default replication in Hadoop). Additionally, we have conducted experiments while varying the hdfs block size to be 64MB, 128MB or 256MB.

6.1.2 Fully Distributed Setup

For the experiments in fully distributed mode, we utilized a cluster containing 5 nodes. Each of the nodes were implemented as virtual machines using VirtualBox [43], a powerful virtualization platform for x86 and intel/AMD64 architectures which is freely available as an Open Source Software to the scientific community under the GNU General Public License (GPL) version 2. All the virtual machines (VMs) are connected via 100Mbps ethernet links enabled by the bridged adapter using PCnet-Fast III (Am79C973) network driver. Table 6.3 shows the configuration of the nodes in this set up. Unlike the pseudo-distributed mode, we have not set any limit on the

number of tasks that each node can run in parallel for each job i.e., there is no limit on how many task execution containers each node can host in parallel for a job. This was done as a way to stress-test our framework during run time. Additionally, we have configured the replication factors for the HDFS blocks for our experiments to be 3, which is the default hdfs replication. To evaluate the performance improvement in heteregenous or shared environments, we configured one of the nodes to be completely dedicated to be the master and it would not run any computation. Similar to the pseudo-distributed mode, we have conducted experiments while varying the hdfs block sizes between 64MB, 128MB and 256MB.

Table 6.7: Configurations of nodes in the cluster

Node ID	**Memory (MB)**	**Disk Space**	**Number of Virtual Processors**
1	4GB	25GB	2
2	2GB	25GB	1
3	2GB	25GB	1
4	2GB	25GB	1
5	2GB	25GB	1

6.2 Evaluation Results

In this section, we present our evaluation results obtained through conducting experiments on the workloads described in Chapter 6.1. We present the results of the experiments on pseudo-distributed setup and fully distributed setup separately and discuss the results corresponding to each setup both separately and together. The metrics chosen for comparison are the percentage of locality improvement for the tasks, task execution times, total job completion time, total tasks killed and performance improvement compared to the default hadoop ecosystem.

6.2.1 Pseudo-distributed Mode

For the pseudo-distributed mode, we mainly looked at the execution time of map phase and reduce phase along with the total job completion time. Figures 6.1 to 6.8 show the results of the experiements that were obtained by running hadoop modified with our framework and the default hadoop over the workloads described in section 6.1. For each of the experiments, we present the total time spent in executing map phase, total time spent in executing reduce phase and the overall completion time of the jobs. Note that the graphs showing map times and reduce times is the total time spent by all maps and reduces (cumulative). Since multiple maps run in parallel, the value of map times and reduce times could be higher than the total job completion time, which is the total time taken for fininshing the job.

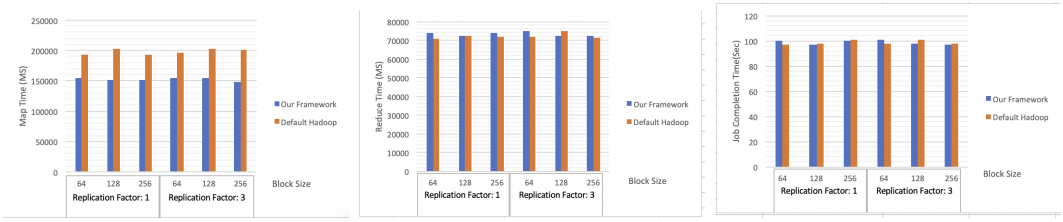

Figure 6.1: GutenbergSmall Workload

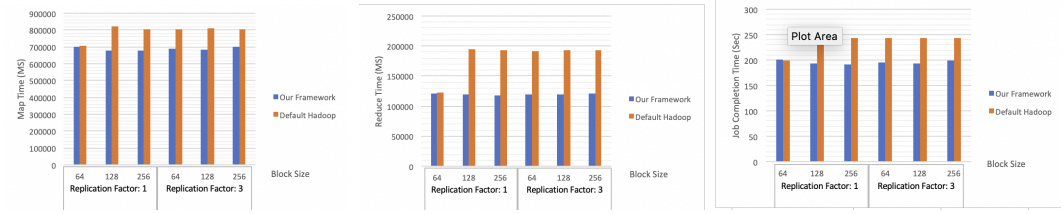

Figure 6.2: GutenbergLarge Workload

From the graphs, it can be observed that the execution times for map and reduce and the overall job completion times are better in most of the cases using our hadoop version compared to the default hadoop. This improvement was present with all three configurations of block sizes.

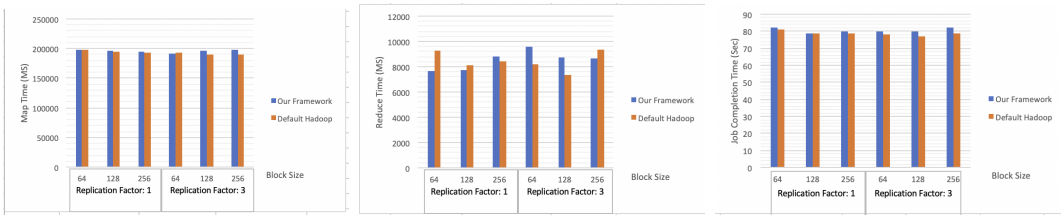

Figure 6.3: BlogCorpusTogether

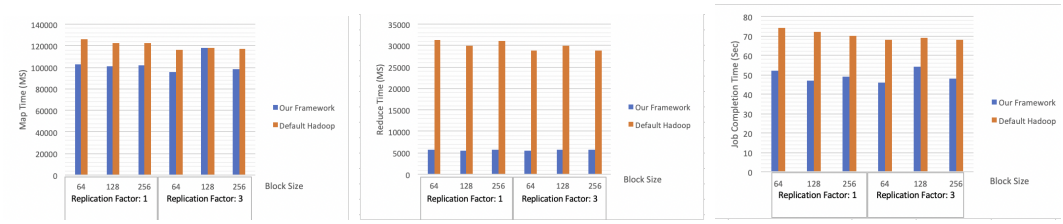

Figure 6.4: BlogCorpus1

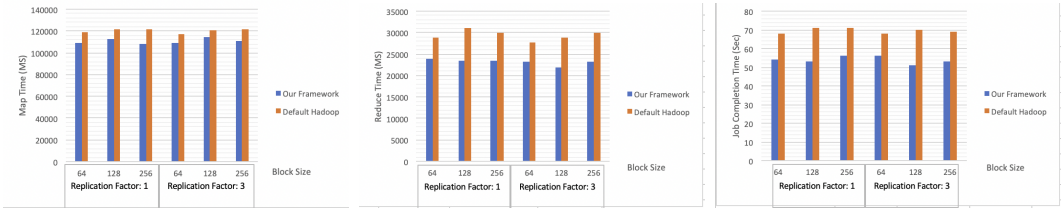

Figure 6.5: BlogCorpus2

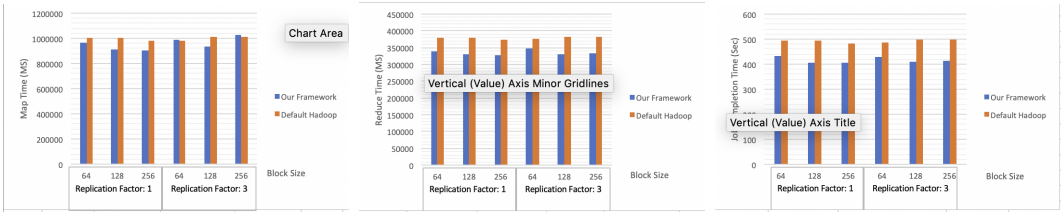

Figure 6.6: 4GBWorkload

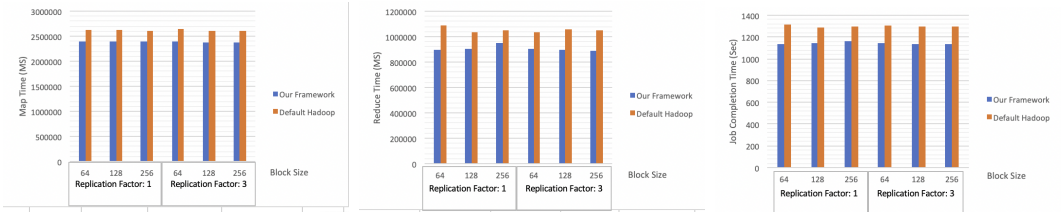

Figure 6.7: 10GBWorkload

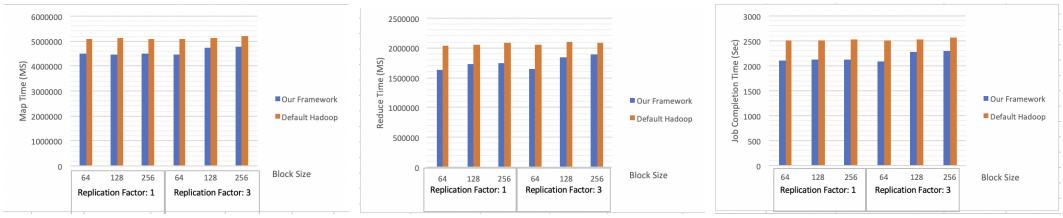

Figure 6.8: 20GBWorkload

The results for the improvements in map times, reduce times and job completion times are summarized in the Table 6.8.

Workload	Input Size	Replication Factor	64MB Block			128MB Block			256 MB Block		
			% improvement in MT	% improvement in RT	% improvement in JCT	% improvement in MT	% improvement in RT	% improvement in JCT	% improvement in MT	% improvement in RT	% improvement in JCT
GutenbergSmall	8MB	1	20.07224264	3.998206717	3	25.43539755	0.084271603	1.020408163	21.96066052	2.600472813	0.99009901
		3	21.00399473	4.248422681	2.97029703	24.04058314	3.141235392	2.97029703	25.62336891	1.486296266	1.020408163
GutenbergLarge	2.61GB	1	0.622388142	1.618956509	0.497512438	17.39842101	38.92039918	21.45748988	15.80347134	39.0440223	20.98765432
		3	14.21893425	37.50201677	19.34156379	15.66053521	38.39668129	20.49180328	12.95836	37.68397294	18.03278689
BlogCorpus1	401.6MB	1	18.28471782	81.90889094	29.72972973	17.28682234	81.29834623	34.72222222	16.32648051	81.8096744	30
		3	17.79327903	80.97984712	32.35294118	0.240845749	80.50108569	21.73913043	15.84099953	80.18068103	29.41176471
BlogCorpus2	404.5MB	1	8.452369968	16.52122519	20.58823529	7.305411811	24.47326816	25.35211268	10.99006227	21.89246161	21.12676056
		3	7.229982964	16.34335073	17.64705882	4.912074447	24.0542692	27.14285714	8.259742832	22.85314406	23.1884058
BlogCorpusCombined	806.2MB	1	10.4549071	92.66448844	32.5	8.516167231	92.619332	33.05785124	14.11994226	92.19007614	36.06557377
		3	12.17739088	92.48047287	35.53719008	12.96048257	91.60872745	34.45378151	10.14963291	91.73126615	34.16666667
BlogCorpusTogether	806.2MB	1	0.577387586	17.73499677	1.219512195	0.338404286	4.63168268	0	1.494903621	4.20416999	1.25
		3	0.3254628	14.38976173	2.5	3.683018099	16.21899646	3.75	4.205645121	6.833297576	3.658536585
10GBWorkload	10.3GB	1	9.250767372	17.81205861	13.44984802	8.645808482	12.63204856	11.45510836	8.202106724	9.642165176	10.20092736
		3	9.172008591	13.05052417	12.13517665	9.187695955	14.98044841	12.6348228	8.663606881	15.1467892	12.44204019
20GBWorkload	20.06GB	1	11.67641832	19.50226684	16.32083001	12.49562614	15.38214637	15.64733916	11.56914969	17.0034483	15.55203799
		3	12.00022493	20.06457333	16.58692185	7.64583637	11.84432043	10.09463722	8.614369058	9.844026629	10.13645224

Table 6.8: *MT- Map Execution Time; RT- Reduce Execution Time; JCT- Job Completion Time*

As you can see our framework was able to provide improvements in map, reduce and job completion times for all the workloads that were included in the experiments. The mean percentage improvement for map time was found to be 9.92, 12.177, 12.558 for block sizes of 64mb, 128mb and 256mb respectively when replication factor was set to 1 and it was found to be 11.7401, 9.7913, 11.7894 for block sizes of 64mb, 128mb and 256mb respectively when replication factor was 3. The mean percentage improvement for reduce time was found to be 31.4701, 33.7426, 33.5483 for block sizes of 64mb, 128mb and 256mb respectively when replication factor was set to 1 and it was found to be 34.882, 35.09322, 33.2199 for block sizes of 64mb, 128mb and 256mb respectively when replication factor was 3. The mean percentage improvement for job completion time was found to be 14.663, 17.839, 17.021 for block sizes of 64mb,

128mb and 256mb respectively when replication factor was set to 1 and it was found to be 17.383, 11.789, 16.507 for block sizes of 64mb, 128mb and 256mb respectively when replication factor was 3. The highest improvement for map time was found for GutenbergSmall workload at 25.62% at a block size of 256mb with a replication factor of 3. The highest improvement for reduce time was found for BlogCorpusCombined workload at 92.66% at a block size of 64MBmb with a replication factor of 1. The highest improvement for job completion time was found for BlogCorpusCombined workload at 36.06% at block size of 256mb with a replication factor of 1.

The lowest improvement for map time was found for BlogCorpus1 workload at 0.2408% at a block size of 128mb with a replication factor of 3. The lowest improvement for reduce time was found for GutenbergSmall workload at 0.0842% at a block size of 128mb with a replication factor of 1. The lowest improvement for job completion time was found for BlogCorpusTogether workload at 0% at block size of 128mb with a replication factor of 1. An interesting point to note is that there was one map task killed while running default hadoop on 10GBWorkload while none was killed when the same workload was executed using our hadoop version. In the case of GutenbergLarge Workload, one map task was killed while running our hadoop version while none was killed when default hadoop was used for that workload. Additionally, one map task was killed for each run on the 20GBWorkload using both our hadoop version and default hadoop. In other words, the distribution of maps killed in our hadoop version and default hadoop seemed to be quite uniform for pseudo-distributed mode. However, the killed maps did not seem to have an effect on the improvements offered by our hadoop version over the default version so we guess that these maps were speculative map tasks [43], which seem to have minimal effect in pseudo-distributed mode.

An interesting observation to mention is that for GutenbergSmall workload, whose

size is 8MB, the improvements in job completion time offered by our hadoop version hovered between 0.9 and 3.9% for varying replication factors and block sizes while improvements in map time were always higher than 20%. Similar numbers in improvements of job completion time were observed when our version was used on workloads whose sizes were around a couple MB similar to that of GutenbergSmall. These were not shown in the table for the sake of brevity. This makes sense because for smaller workloads, the internal overhead produced from Hadoop takes over any improvements offered from the map execution time. The reduce time for the workloads has also improved, which we think is due to better utilization of in-memory blocks during the shuffle and reduce phase. Since in our hadoop version, even during the reduce phase we first try to retain needed data blocks in the memory before evicting them, and in this case spilling them to the disk, this offers significant advantage in accessing those blocks and hence the noticable improvements. But it must be noted that the impact of reduce time over the job completion time is usually minimal as the total time spent in reduce phase is much smaller overall. This is especially true for pseudo-distributed mode where there is only one reducer.

For larger workloads with sizes ranging between 400MB and 20GB, the improvements in job completion time hover between 15 and 34.16%, ignoring one outlier with 0% improvement in job completion time. For the same workloads, the improvements in map execution time hovered between 0.325% and 18.28%, with the workloads at the larger end of the size spectrum (at least 2GB) having their mean map improvements relatively lower than that of the workloads at lower end of the size spectrum (around 400MB). The map improvement for these larger workloads is relatively smaller than the small workloads discussed in the previous paragraph. Despite this, the improvements in job completion time for the larger jobs seem to be higher with our hadoop verison compared to those of the smaller workloads. An important observation which

must be noted for many of these larger workloads is that the percentage improvements in map time is relatively smaller compared to the percentage improvements in the total job completion times i.e., the percentage improvements in job completion time is higher than the percentage improvements in map time. To provide an example of one such situation, one of the runs of BlogCorpus1 with replication factor of 3 and block size of 128mb provides only 0.2408% improvement in map time but the improvement in job completion time was found to be 21.73%. We theorize that higher improvements in job completion time for these larger workloads despite their relatively lower improvements in map time must come from one of the two- i) due to the improvements in reduce times or ii) due to the improvements offered by our hadoop version's algorithms in task selection, container launch, and better memory utilization. But keeping in mind that reduce phase is much smaller and has little effect on the total job completion time as noted in the previous paragraph, thus we think the improvements in job completion time can be attributed to our hadoop version's better task selection, container launch and effective utilization of the in-memory blocks which is significant during the shuffle phase. There is currently no way to collect the exact time taken for container placement, task selection provided by hadoop exactly but the effect our hadoop version has on them can be evidently seen.

6.2.2 Fully Distributed Mode

For the Fully distributed mode, we looked at the execution time of map phase and reduce phase, total job completion time, number of map tasks killed and number of data local tasks vs non-local tasks. Figures 6.9 to 6.17 show the results of the experiements that were obtained by running our version of hadoop and the default hadoop over the workloads described in the Chapter 6.1. The only newer workloads chosen for fully

distributed mode are GutenSmallA, GutenSmallB, BlogCorpusTogether and 2GB-Workload which contain 29, 31, 2 and 1 input files each with cumulative sizes of 3.7MB, 4.3MB, 806.1MB and 2.27 GB respectively. It must be noted that for the experiments in fully distributed mode, only replication factor of 3 was chosen since this is the most common and default replication factor used in hadoop clusters. The reason we chose replication factors of both 1 and 3 for the experiments conducted in pseudo-distributed mode is for the sake of sanity check on our framework. Similar to the experiments presented in the pseudo-distributed mode, the map time and reduce time is the total time spent by all maps and reduces (cumulative). Since multiple maps run in parallel, the value of map time and reduce time could be higher than the total job completion time, which is the total time taken for fininshing the job.

Additionally, we did not include the number of reduce tasks killed in our results because we have found very minimal number of reduce tasks killed in both cases (just one occurrence among all the experiments).

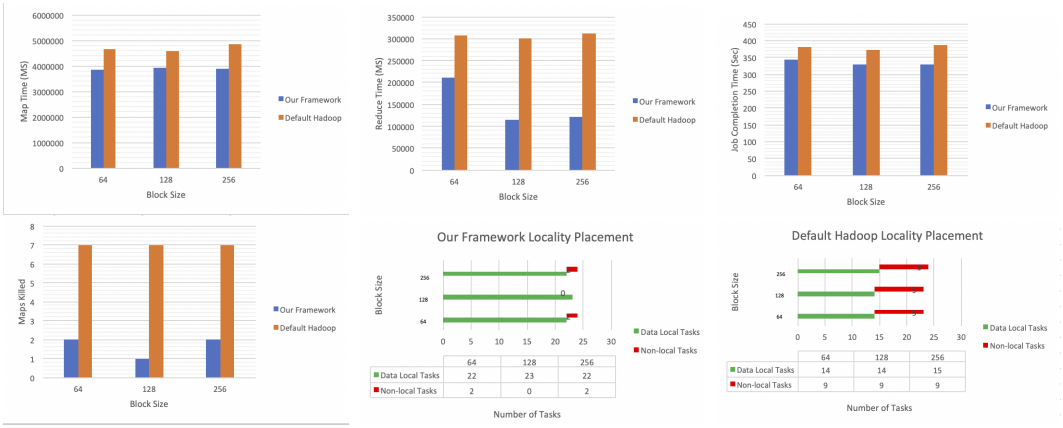

Figure 6.9: GutenbergLarge Workload

The results for the improvements in map time, reduce time, job completion time are summarized in Tables 6.9, 6.10 and 6.11 for block sizes 64 MB, 128 MB and 256 MB respectively.

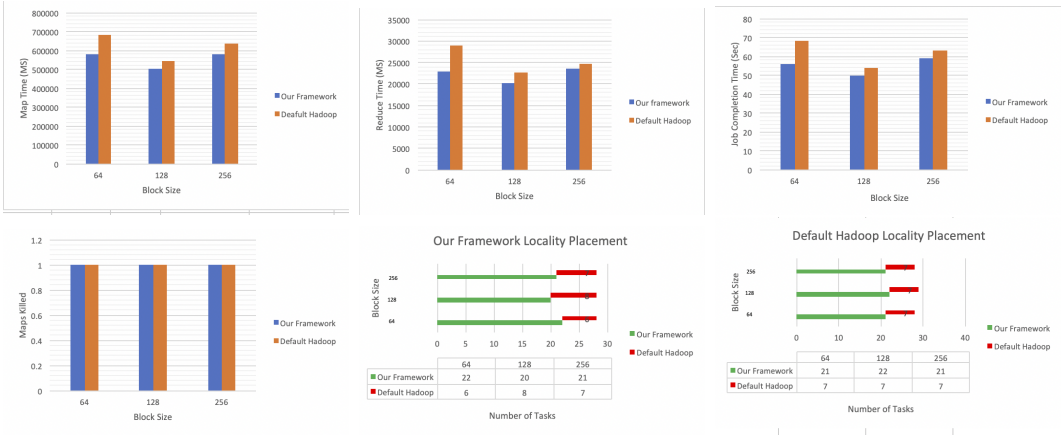

Figure 6.10: GutenbergSmallA Workload

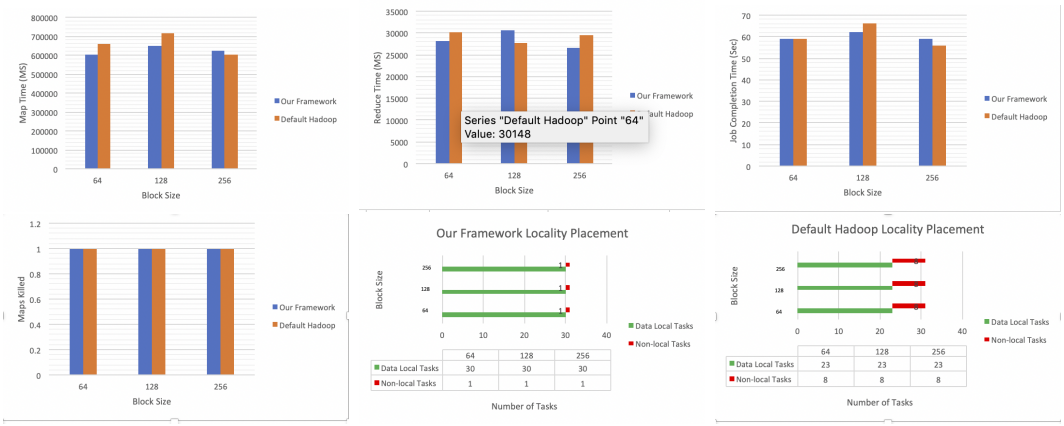

Figure 6.11: GutenbergSmallB Workload

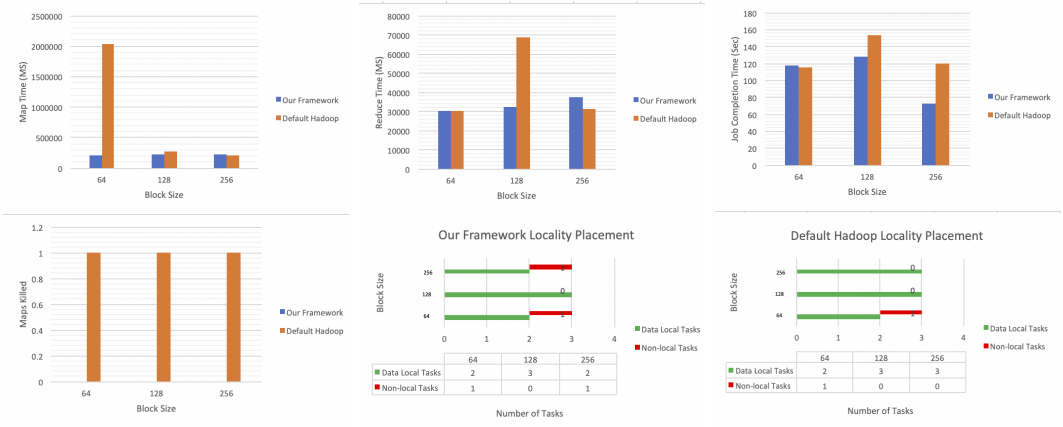

Figure 6.12: BlogCorpus1 Workload

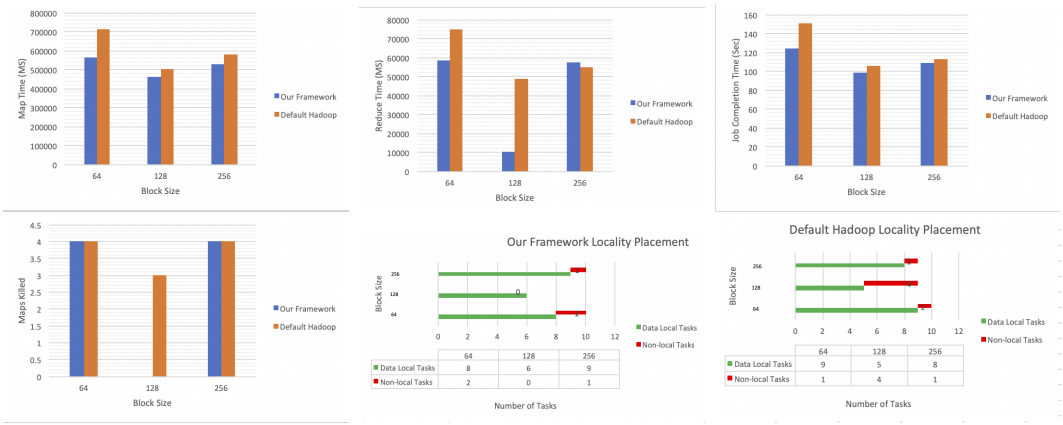

Figure 6.13: BlogCorpus2 Workload

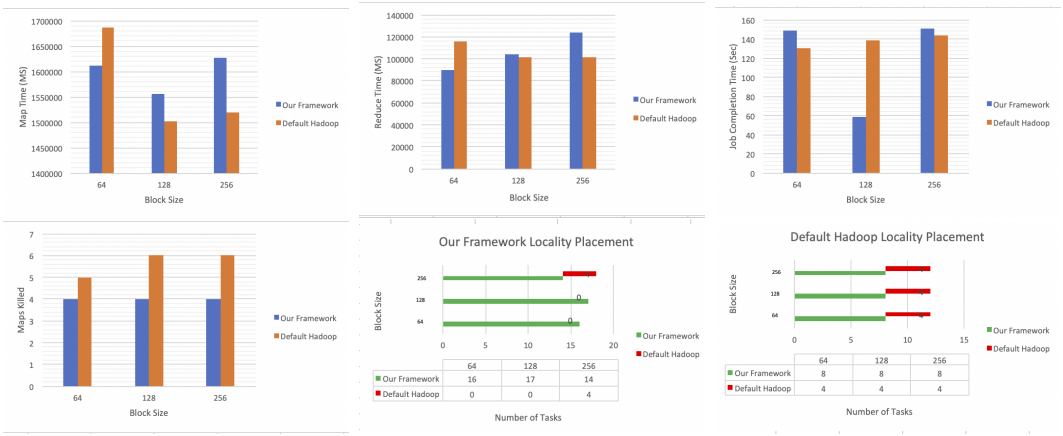

Figure 6.14: BlogCorpusCombined Workload

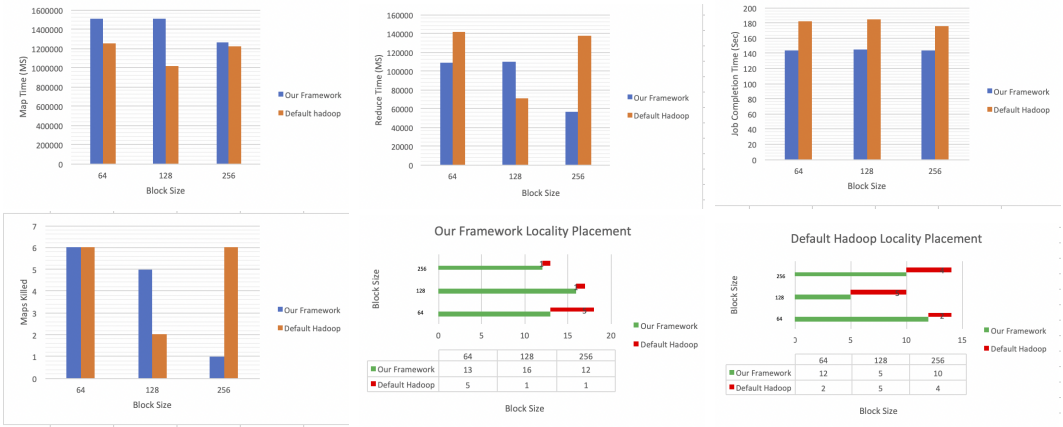

Figure 6.15: BlogCorpusTogether Workload

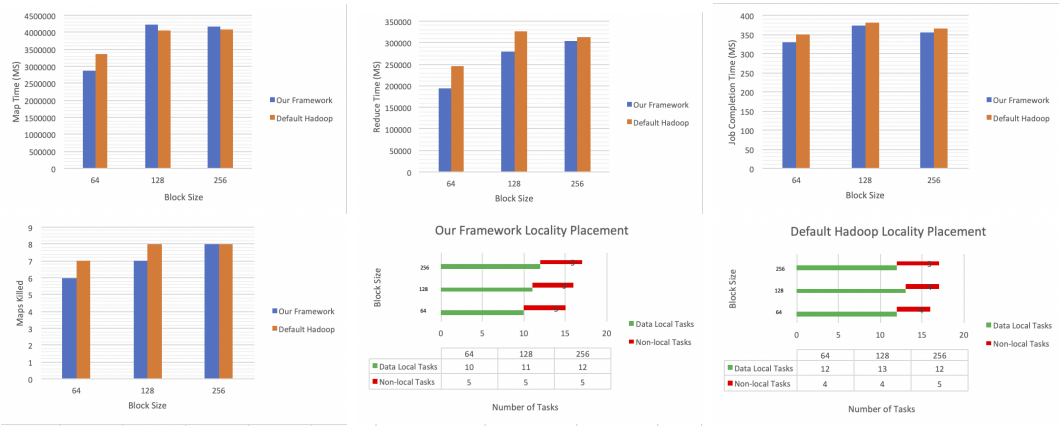

Figure 6.16: 2GB Workload

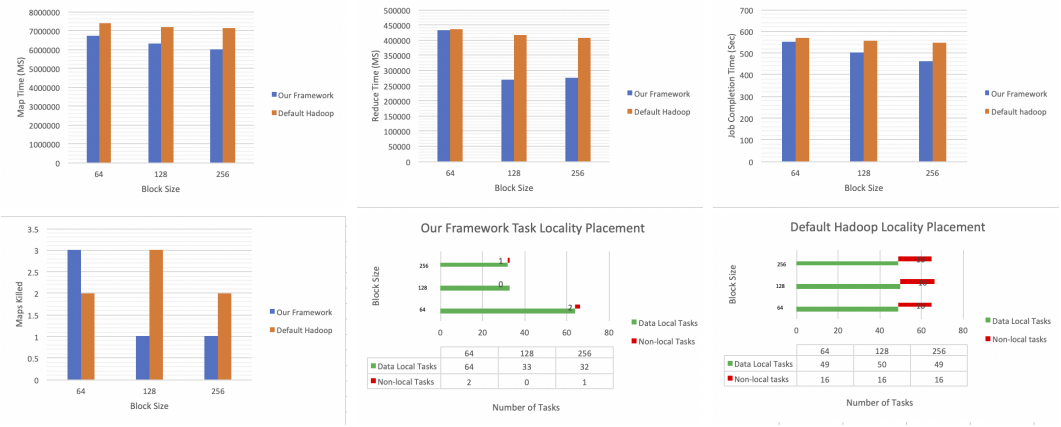

Figure 6.17: 4GB Workload

Workload	Input Size	Replication Factor	64MB Block		
			% improvement in MT	% improvement in RT	% improvement in JCT
GutenbergSmallA	3.7MB	3	15.15575036	21.28482171	21.42857143
GutenbergSmallB	4.3MB	3	9.215805177	6.438238026	0
GutenbergLarge	2.61GB	3	17.49806978	31.57465502	10
BlogCorpusTogether	806.2MB	3	-19.91835459	22.66314245	20.87912088
BlogCorpusCombined	806.2MB	3	4.348290326	22.40008316	-14.61538462
BlogCorpus2	404.5MB	3	20.87614045	21.86157582	17.8807947
BlogCorpus1	401.6MB	3	89.92036608	-0.428576138	-1.724137931
2GBWorkload	2.27GB	3	14.24189306	21.26486588	6
2GBWorkload	4.23GB	3	9.13471213	0.690377336	3.321678322

Table 6.9: *MT- Map Execution Time; RT- Reduce Execution Time; JCT- Job Completion Time*

Workload	Input Size	Replication Factor	128MB Block		
			% improvement in MT	% improvement in RT	% improvement in JCT
GutenbergSmallA	3.7MB	3	7.182342839	10.83717864	8
GutenbergSmallB	4.3MB	3	9.57210251	-10.66368651	6.060606061
GutenbergLarge	2.61GB	3	14.32080506	62.23988358	11.55913978
BlogCorpusTogether	806.2MB	3	-47.91331411	-55.9947896	21.62162162
BlogCorpusCombined	806.2MB	3	-3.595430936	-2.734251345	57.24637681
BlogCorpus2	404.5MB	3	8.210546859	78.5777558	7.547169811
BlogCorpus1	401.6MB	3	19.47432295	53.04700681	16.33986928
2GBWorkload	2.27GB	3	-4.2861816	14.2909161	1.578947368
2GBWorkload	4.23GB	3	11.68967532	35.17485222	10.21505376

Table 6.10: *MT- Map Execution Time; RT- Reduce Execution Time; JCT- Job Completion Time*

Workload	Input Size	Replication Factor	256 MB Block		
			% improvement in MT	% improvement in RT	% improvement in JCT
GutenbergSmallA	3.7MB	3	8.154249109	4.377104377	6.779661017
GutenbergSmallB	4.3MB	3	-3.306340546	9.199115797	-5.357142857
GutenbergLarge	2.61GB	3	19.87530912	60.92918792	14.72868217
BlogCorpusTogether	806.2MB	3	-3.473782065	58.98028779	18.75
BlogCorpusCombined	806.2MB	3	-7.102138517	-21.88228811	-4.861111111
BlogCorpus2	404.5MB	3	8.910972971	-4.358362334	3.539823009
BlogCorpus1	401.6MB	3	-9.854472252	-18.48621809	39.16666667
2GBWorkload	2.27GB	3	-1.983159506	3.06137746	3.01369863
2GBWorkload	4.23GB	3	16.12840045	32.10721224	15.48269581

Table 6.11: *MT- Map Execution Time; RT- Reduce Execution Time; JCT- Job Completion Time*

As you can see our framework was able to provide improvements in map, reduce

and job completion time in most of the experiments. To make it easier to distinguish, the experiments where our version was not able to improve the map time, reduce time and job completion time are colored in red in Tables 6.9, 6.10 and 6.11. The mean percentage improvement for map time was found to be 18.9172249%, 0.37064918%, 1.4025798% for block size of 64mb, 128mb and 256mb respectively. The mean percentage improvement for reduce time was found to be 18.38235%, 18.7%, 11.4775256% for block size of 64mb, 128mb and 256mb respectively. The mean percentage improvement for job completion time was found to be 7.48112055%, 16.2442163%, 9.47003% for block size of 64mb, 128mb and 256mb respectively. The highest improvement for map time was found for BlogCorpus1 workload at 89.92% at a block size of 64mb. The highest improvement for reduce time was found for BlogCorpus2 workload at 78.577% at a block size of 128mb. The highest improvement for job completion time was found for BlogCorpusCombined workload at 57.246% at block size of 128mb.

There were some experiments where the default hadoop fared better than our version. These are colored in red in the Table 6.9. The lowest improvement for map time was found for BlogCorpusTogether workload at -47.91331411% at a block size of 128mb. The lowest improvement for reduce time was found for BlogCorpusTogether workload at -55.9947896% at a block size of 128mb. The lowest improvement for job completion time was found for BlogCorpusCombined workload at -14.61538462% at block size of 64mb.

An interesting observation to mention is that for experiments where our version of hadoop fared worse in map time and/or reduce times, it still fared much better in improving the total job completion time for that same experiment. An example of this situation is the experiment on BlogCorpusTogether workload, where the map time fared worse using our hadoop version for all three block sizes that were included in the experiments. However, the total job completion time improved by a mean of

20.4169%. Second example for this can be observed from the experiments on the BlogCorpus1 workload where the map time and reduce time fared worse when the block size was set to 256mb but the total job completion time still improved by 39.16%. The same situation occured for all experiments where the map and reduce time were slightly worse while the total job completion times still fared much better. The only exception to this is the experiments on the BlogCorpusCombined, where the total job completion time fared worse using our hadoop version for block size of 64mb and 256mb. it must also be noted for the same workload, total job completion time improved by 57.246% when block size was set to 128mb. Overall, when we consider the mean improvements in job completion times all the workloads fared better using our hadoop version.

It must also be mentioned that for smaller workloads, our hadoop version fared better in improving map, reduce and total job completion time. This is interesting because, for some of the experiments on the smaller workloads in the pseudo-distributed mode, map and reduce time were better using default hadoop even though the total job completion time were better using our hadoop version. This situation did not repeat for fully distributed mode- our hadoop version fared better in improving all three metrics -map, reduce and total job completion time as seen from the graphs and tables. This means that our hadoop version not only reduced the internal overhead caused from hadoop but also improved the task execution itself.

Additionally, we theorize that the improvemnts in reduce and map time are due to better utilization of in-memory blocks while the total job completion time improvements can be attributed to both i) improvments from map and reduce time due to better utilization of memory ii) better task and conatiner placement decisions made by our scheduler algorithm described in Chapter 4. Since in our hadoop version, even during the reduce phase we first try to retain needed data blocks in the memory

before evicting them, and in this case spilling them to the disk, this offers significant advantage in accessing those blocks and hence the noticable improvements. A small note that must be mentioned is that similar to the pseudo-distributed mode, the impact of reduce times over the job completion time is usually minimal as the total time spent in reduce phase is much smaller overall.

Table 6.12 shows the side-by-side comparison of the total map tasks killed for both our hadoop version and default version.

Workload	Input Size	Replication Factor	64MB Block		128MB Block		256 MB Block	
			Our Framework	Default Hadoop	Our Framework	Default Hadoop	Our Framework	Default Hadoop
GutenbergSmallA	3.7MB	3	1	1	1	1	1	1
GutenbergSmallB	4.3MB	3	1	1	1	1	1	1
GutenbergLarge	2.61GB	3	2	7	1	7	2	7
BlogCorpusTogether	806.2MB	3	6	6	5	2	1	6
BlogCorpusCombined	806.2MB	3	4	5	4	6	4	6
BlogCorpus1	401.6MB	3	0	1	0	1	0	1
BlogCorpus2	404.5MB	3	4	4	0	3	4	4
2GBWorkload	2.27GB	3	6	7	7	8	8	8
4GBWorkload	4.23GB	3	3	2	1	3	1	2

Table 6.12: *The total number of map tasks killed. Note that we do not present the number of reduce tasks killed in our results becaue we have found very minimal number of reduce tasks killed, that is in both cases there is just one occurrence among all the experiments.*

As Table 6.12 shows, the number of map tasks killed were less using our hadoop version in 52 of 54 experiments that were conducted. That means in 96.3% of the experiments, our hadoop version had less number of map tasks killed compared to default hadoop. First let's discuss why map tasks or tasks in general in hadoop can be launched only to be killed. Since tasks in hadoop run in parallel across several machines, some of the tasks can be slower compared to rest of the tasks. The slow tasks can occur due to several reasons like insufficient resources, hardware issues, improper utilization of resources like memory and CPU etc. These slow tasks can still end up

finishing their execution but they will slow down the overall progress of the job. The approach taken by hadoop to address the latency added to the job execution due to such slow tasks is to launch duplicate tasks in parallel on nodes that have shown to execute tasks relatively faster. This way it ensures that it has the duplicate tasks to fall back on in case the slow tasks start to become an issue. This technique is called speculative task exceution in hadoop. Hadoop kills whichever task that has not been finished timely, which could be the original task or the newly launched speculative task *i.e.,* the one that does not finish before the other. Speculative execution of tasks is enabled by default in hadoop and can be configured by the parameter mapreduce.map.speculative and mapreduce.reduce.speculative . As the results show our hadoop version shows less maps killed over almost all the cases. The only exceptions are two cases which are colored in red in Table 6.12. We theorize that our hadoop version fares much better in number of maps killed because of the overall improvement it offers in the execution of tasks. Since the tasks are executed faster, hadoop does not see as many slower tasks and hence it does not launch as many speculative tasks which will result in higher number of killed tasks overall. This is a testament to the improvements in performance offered by our hadoop version since higher number of speculative tasks and hence higher number of killed tasks signals relatively lower performance of the system. It must be noted that we were not expecting to see this positive result in the number of killed tasks but only noticed this while going through the collected data from the experiments.

The results for the percentage of locality placement of the tasks are summarized in Table 6.13. Table 6.13 shows the side-by-side comparison of the locality placement of the tasks for our hadoop version vs default hadoop. The table does not show the percentage improvements in locality rather it shows for both the percentage of the data local tasks i.e. the percentage of the tasks that were placed on nodes with data

on them. The results colored in red are the ones where default hadoop fared better in achieving higher locality of the tasks. It must be noted that the non-local tasks in our experiments are basically rack-local tasks, there are no off-rack tasks because of the organization of our virtual cluster used for the exprimentation.

Workload	Input Size	Replication Factor	64MB Block		128MB Block		256 MB Block	
			Our Framework	Default Hadoop	Our Framework	Default Hadoop	Our Framework	Default Hadoop
GutenbergSmallA	3.7MB	3	78.57142857	75	71.42857143	75.86206897	75	75
GutenbergSmallB	4.3MB	3	96.77419355	74.19354839	96.77419355	74.19354839	96.77419355	74.19354839
GutenbergLarge	2.61GB	3	91.66666667	60.86956522	100	60.86956522	91.66666667	62.5
BlogCorpusTogether	806.2MB	3	72.22222222	85.71428571	94.11764706	50	92.30769231	71.42857143
BlogCorpusCombined	806.2MB	3	100	66.66666667	100	66.66666667	77.77777778	66.66666667
BlogCorpus2	404.5MB	3	80	90	100	55.55555556	90	88.88888889
BlogCorpus1	401.6MB	3	66.66666667	66.66666667	100	100	66.66666667	100
2GBWorkload	2.27GB	3	66.66666667	75	68.75	76.47058824	70.58823529	70.58823529
4GBWorkload	4.23GB	3	96.96969697	75.38461538	100	75.75757576	96.96969697	75.38461538

Table 6.13: Table shows the total percentage of local tasks.

As the results show, our hadoop version achieves much higher locality placement of the tasks. 49 of 54 experiments show that our hadoop version places more local tasks compared to the default hadoop. Achieving higher locality in task scheduling has been considered to be a prime metric in improving job performance and over all cluster performance for quite a long time. It must be noted that in the cases where default hadoop was able to achieve higher locality placement, the locality offered by our hadoop version was still within a margin of 5-11% difference in almost all the cases. The above results corroborate the improvements offered by our strategies, task selection and container assignment.

Chapter 7

Conclusion and Future Work

In our work, we have presented a scheduling and data prefetching framework as part of hadoop that leverages the data access patterns across the cluster and the centralized memroy awareness to improve the overall system performance. Our framework leverages centralized knowledge of the data access patterns of cluster node's memories to make better locality based task assignment decisions as well as prefetching decisions. Additionally, our framework includes a couple of data management algorithms specifically for the memory of the nodes in the cluster. These algorithms take the responsibility of performing the eviction and retention decisions of the memory blocks based on data access metrics. On this end, we have presented couple of metrics that allow us to achieve our stated goals. We have developed our framework as part of Hadoop 2.8 ecosystem and have made it publicly available via GitHub. We have then corroborated our framework through the comparison of experimental results on the well-known WordCount benchmark using both pseduo-distributed cluster and fully distributed cluster against the standard hadoop system. We have presented the results which show that compared to default hadoop system, our framework achieves improvements in 1) locality 2) faster task execution time, 3) faster job completion

time and 4) less number of killed tasks.

Now, we discuss some of the aspects of our framework that will be part of the future work as well as some of the limitations to our approach. We plan to test our framework on much larger workloads with sizes ranging from couple of hundred GB to further corroborate our approach as well as offer a solid form of stress testing for our framework's effectiveness. Additionally, due to resource constraints the experiments were only conducted in a small sized cluster consisting of a few nodes. We plan to continue experimentation to see how our framework performs on a much larger cluster consisting of anywhere between 20 to hundreds of nodes. A limitation in our experiments is that the cluster had no off-switch nodes. We theorize that this fact does not affect the locality improvements offered by our framework because a data local task by definition is a task placed on node containing its data and hence the presence or absence of off-switch nodes should not considerably affect our scheduler's locality placement all things being ideal. Also, keeping in mind that hadoop in general performs better when there are small number of large workloads rather than large number of small workloads, we would like to see how our approach fares under such a condition. It must be noted that such a condition occurs occcasionally in large heteregenous clusters in real-world. We would also like to see which aspects of our framework are repsonsible for the improvements across various metrics like map time, job completion time etc. For example, we would like to see what percentage of the improvement seen in map execution time is directly attributed due to the scheduler in our framework. It is easy to directly attribute other improvements, for example, the locality placement to the scheudler in our framework but others areas are not so clear. Currently, there is no existing way to exactly quantify such contribution to differeent parts of our framework but that will be part of our future work. Another aspect to improve upon is the presence of couple of outliers in our experimental results, where

for a particular experiment on a particular metrics like, for example, map time, our framework fared relatively worse. Although such cases occurred occasionally, we have no explanation currently as to why they occured. Finally, we would like to quantify what contribution the memory load balancing scheme provides towards overall task and job completion speed ups that were shown in our results.

Bibliography

[1] Dean, J. and Ghemawat, S., 2008. MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1), pp.107-113.

[2] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). Spark: Cluster computing with working sets. HotCloud, 10(10-10), 95. Chicago

[3] Li, S., Hu, S., Wang, S., Su, L., Abdelzaher, T., Gupta, I., & Pace, R. (2014, June). Woha: Deadline-aware map-reduce workflow scheduling framework over hadoop clusters. In 2014 IEEE 34th International Conference on Distributed Computing Systems (pp. 93-103). IEEE. Chicago

[4] Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., & Stoica, I. (2010, April). Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In Proceedings of the 5th European conference on Computer systems (pp. 265-278). ACM.

[5] Ananthanarayanan, G., Ghodsi, A., Shenker, S., & Stoica, I. (2011, May). Disk-locality in datacenter computing considered irrelevant. In HotOS (Vol. 13, pp. 12-12).

[6] Guo, Z., Fox, G., & Zhou, M. (2012, May). Investigation of data locality in mapreduce. In Proceedings of the 2012 12th IEEE/ACM International Symposium on

Cluster, Cloud and Grid Computing (ccgrid 2012) (pp. 419-426). IEEE Computer Society.

[7] Phan, L. T., Zhang, Z., Zheng, Q., Loo, B. T., & Lee, I. (2011, December). An empirical analysis of scheduling techniques for real-time cloud-based data processing. In 2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA) (pp. 1-8). IEEE.

[8] Tang, Z., Zhou, J., Li, K., & Li, R. (2013). A MapReduce task scheduling algorithm for deadline constraints. Cluster computing, 16(4), 651-662. Chicago

[9] Hwang, J., Uppal, A., Wood, T., & Huang, H. (2014). Mortar: Filling the gaps in data center memory. ACM SIGPLAN Notices, 49(7), 53-64.

[10] Ananthanarayanan, G., Ghodsi, A., Warfield, A., Borthakur, D., Kandula, S., Shenker, S., & Stoica, I. (2012). Pacman: Coordinated memory caching for parallel jobs. In Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12) (pp. 267-280). Chicago

[11] Xie, J. (2012). Improving Performance of Hadoop Clusters (Doctoral dissertation).

[12] Sun, M., Zhuang, H., Zhou, X., Lu, K., & Li, C. (2014, August). HPSO: Prefetching based scheduling to improve data locality for MapReduce clusters. In International Conference on Algorithms and Architectures for Parallel Processing (pp. 82-95). Springer, Cham.

[13] Seo, S., Jang, I., Woo, K., Kim, I., Kim, J. S., & Maeng, S. (2009, August). Hpmr: Prefetching and pre-shuffling in shared mapreduce computation environment. In

2009 IEEE International Conference on Cluster Computing and Workshops (pp. 1-8). IEEE.

[14] Ananthanarayanan, G., Agarwal, S., Kandula, S., Greenberg, A., Stoica, I., Harlan, D., & Harris, E. (2011, April). Scarlett: coping with skewed content popularity in mapreduce clusters. In Proceedings of the sixth conference on Computer systems (pp. 287-300). ACM. Chicago

[15] Abad, C. L., Lu, Y., & Campbell, R. H. (2011, September). DARE: Adaptive data replication for efficient cluster scheduling. In 2011 IEEE international conference on cluster computing (pp. 159-168). Ieee. Chicago

[16] Ibrahim, S., Jin, H., Lu, L., He, B., Antoniu, G., & Wu, S. (2012, May). Maestro: Replica-aware map scheduling for mapreduce. In Proceedings of the 2012 12th ieee/acm international symposium on cluster, cloud and grid computing (ccgrid 2012) (pp. 435-442). IEEE Computer Society.

[17] Gu, T., Zuo, C., Liao, Q., Yang, Y., & Li, T. (2013). Improving MapReduce performance by data prefetching in heterogeneous or shared environments. International Journal of Grid & Distributed Computing, 6(5), 71-82. Chicago

[18] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., ... & Saha, B. (2013, October). Apache hadoop yarn: Yet another resource negotiator. In Proceedings of the 4th annual Symposium on Cloud Computing (p. 5). ACM. Chicago

[19] Hadoop, Apache. "Hadoop." 2009-03-06]. http://hadoop. apache. org (2009).

[20] Shvachko, K., Kuang, H., Radia, S., & Chansler, R. (2010, May). The hadoop distributed file system. In 2010 IEEE 26th symposium on mass storage systems and technologies (MSST) (pp. 1-10). Ieee. Chicago

[21] Wang, W., & Ying, L. (2014, September). Data locality in MapReduce: A network perspective. In 2014 52nd Annual Allerton Conference on Communication, Control, and Computing (Allerton) (pp. 1110-1117). IEEE. Chicago.

[22] Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J.M., Ramasamy, K. and Taneja, S., 2015, May. Twitter heron: Stream processing at scale. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (pp. 239-250). ACM.

[23] Iqbal, M. H., & Soomro, T. R. (2015). Big data analysis: Apache storm perspective. International journal of computer trends and technology, 19(1), 9-14. Chicago

[24] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S. and Tzoumas, K., 2015. Apache flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 36(4).

[25] Blumofe, R.D. and Leiserson, C.E., 1999. Scheduling multithreaded computations by work stealing. Journal of the ACM (JACM), 46(5), pp.720-748.

[26] Guo, Z. and Fox, G., 2012, May. Improving mapreduce performance in heterogeneous network environments and resource utilization. In 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012) (pp. 714-716). IEEE.

[27] Silva, R.F.E. and Carpenter, P.M., 2016, November. Controlling network latency in mixed hadoop clusters: Do we need active queue management?. In 2016 IEEE 41st Conference on Local Computer Networks (LCN) (pp. 415-423). IEEE.

[28] Rista C, Griebler D, Maron CA, Fernandes LG. Improving the network performance of a container-based cloud environment for hadoop systems. In2017 International Conference on High Performance Computing & Simulation (HPCS) 2017 Jul 17 (pp. 619-626). IEEE.

[29] Apache Software Foundation. [Online]. http://www.apache.org/

[30] NameNode. https://wiki.apache.org/hadoop/NameNode

[31] Wang, W., Zhu, K., Ying, L., Tan, J., & Zhang, L. (2016). Maptask scheduling in mapreduce with data locality: Throughput and heavy-traffic optimality. IEEE/ACM Transactions on Networking (TON), 24(1), 190-203.

[32] Abad, C. L., Lu, Y., & Campbell, R. H. (2011, September). DARE: Adaptive data replication for efficient cluster scheduling. In 2011 IEEE international conference on cluster computing (pp. 159-168). Ieee.

[33] Li, C., Zhang, J., Chen, Y., & Luo, Y. (2019). Data Prefetching and File Synchronizing for Performance Optimization in Hadoop-Based Hybrid Cloud. Journal of Systems and Software. Chicago

[34] Palanisamy, B., Singh, A., Liu, L., & Jain, B. (2011, November). Purlieus: locality-aware resource allocation for MapReduce in a cloud. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (p. 58). ACM.

[35] Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., & Goldberg, A. (2009, October). Quincy: fair scheduling for distributed computing clusters. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (pp. 261-276). ACM.

[36] Li, C., Zhang, J., Ma, T., Tang, H., Zhang, L., & Luo, Y. (2019). Data locality optimization based on data migration and hotspots prediction in geo-distributed cloud environment. Knowledge-Based Systems, 165, 321-334.

[37] Choi, D., Jeon, M., Kim, N., & Lee, B. D. (2017). An enhanced data-locality-aware task scheduling algorithm for hadoop applications. IEEE Systems Journal, (99), 1-12.

[38] He, C., Lu, Y., & Swanson, D. (2011, November). Matchmaking: A new mapreduce scheduling technique. In 2011 IEEE Third International Conference on Cloud Computing Technology and Science (pp. 40-47). IEEE.

[39] Zhang, X., Zhong, Z., Feng, S., Tu, B., & Fan, J. (2011, May). Improving data locality of mapreduce by scheduling in homogeneous computing environments. In 2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications (pp. 120-126). IEEE.

[40] Gutenberg Project, http://www.gutenberg.org

[41] J. Schler, M. Koppel, S. Argamon and J. Pennebaker (2006). Effects of Age and Gender on Blogging in Proceedings of 2006 AAAI Spring Symposium on Computational Approaches for Analyzing Weblogs.

[42] VirtualBox, https://www.virtualbox.org

[43] Chen, Q., Liu, C. and Xiao, Z., 2014. Improving MapReduce performance using smart speculative execution strategy. IEEE Transactions on Computers, 63(4), pp.954-967.