Efficient Data Access Strategies for Hadoop and Spark on HPC Cluster with Heterogeneous Storage*

Nusrat Sharmin Islam, Md. Wasi-ur-Rahman, Xiaoyi Lu, and Dhabaleswar K. (DK) Panda
Department of Computer Science and Engineering, The Ohio State University
{*islamn, rahmanmd, luxi, panda*}*@cse.ohio-state.edu*

***Abstract*—The most popular Big Data processing frameworks of these days are Hadoop MapReduce and Spark. Hadoop Distributed File System (HDFS) is the primary storage for these frameworks. Big Data frameworks like Hadoop MapReduce and Spark launch tasks based on data locality. In the presence of heterogeneous storage devices, when different nodes have different storage characteristics, only locality-aware data access cannot always guarantee optimal performance. Rather, storage type becomes important, specially when high performance SSD and in-memory storage devices along with high performance interconnects are available. Therefore, in this paper, we propose efficient data access strategies (e.g. Greedy (prioritizes storage type over locality), Hybrid (balances the load for locality and high performance storage), etc.) for Hadoop and Spark considering both data locality and storage types. We re-design HDFS to accommodate the enhanced access strategies. Our evaluations show that, the proposed data access strategies can improve the read performance of HDFS by up to 33% compared to the default locality-aware data access. The execution times of Hadoop and Spark Sort are also reduced by up to 32% and 17%. The performances of Hadoop and Spark TeraSort are also improved by up to 11% through our design.**

I. Introduction

Data generation and analytics have taken a massive leap in recent times due to the continuous growth of popularity for social gaming and geospatial applications. According to [10], at any given hour, around ten million people are logged into their favorite social game, like Candy Crush Saga from King [6] or the Clash of Clans from Supercell [8]. The more recent craze of Pokemon GO has even emphasized the inevitability of data growth and the necessity of faster processing with supreme storage requirements - in other words, technological advancements for Big Data middleware, in the near future. According to a recent Datanami article [11], people are now spending twice as much time playing Pokemon GO as they are in Facebook. It also mentioned that the social media traffic from Twitter, Snapchat, and YouTube are going down because of the massive level usage of Pokemon GO. This phenomenon is presenting significant challenges not only to manage, store, and protect the sheer volume and diversity of the data, but also to operate on this data to perform large scale data analysis.

MapReduce [13] has provided a major breakthrough through its scalable design that provides parallel programmability in addition to fault tolerance. Since its inception, the open-source version of MapReduce, namely Hadoop [30], has been utilized for data processing with large-capacity storage. It has also paved the way for in-memory based data analytics engine, such as Spark [34]. These middleware are being widely used in enterprise data centers and modern High-Performance Computing (HPC) clusters. For data storage functionalities of these processing engines, distributed file systems such as HDFS [30] are the most popular. To enable large scale data processing with Hadoop and Spark, fast I/O support is one of the basic necessities. Recent studies [18, 26, 27] have demonstrated that the bulk I/O write operation in these Big Data middleware is one of the major performance bottlenecks. In addition to this, data access performance also needs to be enhanced to attain significant improvement in overall application performance. Also, as modern HPC clusters are evolving, there has been a fundamental change in the type of data storage in these systems. HPC clusters are providing fast high-performance storage in addition to the traditional high-capacity storage separated out from the compute engine through high-performance interconnects, such as InfiniBand. All these changes in the modern HPC systems must be handled technologically to provide fast data storage for Big Data processing.

A. Motivation

HPC clusters are equipped with heterogeneous storage devices and high performance interconnects. As a result, different nodes in the cluster have different storage characteristics. Figure 1 shows an example of HPC cluster with heterogeneous storage characteristics and how HDFS is deployed in this setup with the DataNodes being equipped with different types of storage devices.

Hadoop MapReduce and Spark use HDFS as the underlying file system. Both of these frameworks read data from HDFS while executing jobs. By default, the schedular launches the tasks considering data locality in order to minimize data movements. The launched tasks read the

*This research is supported in part by National Science Foundation grants #CNS-1419123, #IIS-1447804, #CNS-1513120, #CCF-1565414, and #IIS-1636846. It used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.

978-1-4673-9005-7/16/$31.00 ©2016 IEEE

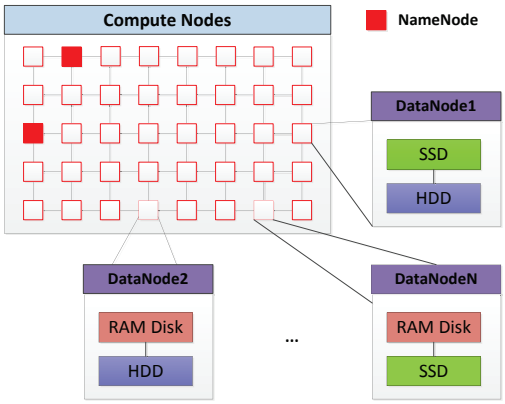


Figure 1: HDFS deployment on HPC cluster with Heterogeneous Storage

data in HDFS without considering the storage type or bandwidth of the storage devices on the the local node. Figure 2 shows the default data access scheme by Hadoop and Spark in HDFS. For most of the commodity clusters, this makes perfect sense as the underlying interconnect is supposed to be slow and hence, data movement is expensive. Besides, HDFS was initially designed for homegeneous systems and the assumption was that the nodes will have hard disks as the storage devices [30]. Recently, the concept of heterogeneous storage devices have been introduced in HDFS. Consequently, a variety of data placement policies have been proposed in the literature [14, 17] to utilize the heterogeneous storage devices efficiently. Even though, with heterogeneous storage devices, data is replicated in different types of storage devices across different nodes, while accessing the data only locality is considered in the default HDFS distributions. As a result, applications and frameworks running over HDFS cannot utilize the hardware capabilities and obtain peak performance in the presence of many concurrent tasks and jobs. Moreover, HPC clusters host heterogeneous nodes with different types of storage devices. The interconnect also provides low latency and high throughput communication. Recent studies [15, 18, 22–25] have demonstrated the performance improvement of different Big Data middleware by taking advantage of HPC technologies. As an example, Triple-H [17, 19] leverages the benefits from Remote Direct Memory Access (RDMA) over InfiniBand and heterogeneous storage to deliver better performance for Hadoop and Spark workloads.

Recently, Cloudera also stressed the importance of storage types while reading data from HDFS [2]. The community has also been talking about locality- and caching-aware task scheduling for both Hadoop and Spark [3, 4]. But Big Data frameworks like Hadoop and Spark can use a variety of schedulers like, Mesos [1], YARN [31], etc. to run their applications. In order to add the support of storage types as well as locality, each of these schedulers has to be modified. On the other hand, most of the Big Data middleware use HDFS as the underlying file system. As a result, if the concept of storage types can be introduced (in addition to data locality) while reading data from HDFS, it saves a lot of efforts by the framework programmer. It is, therefore, critical to rethink the data access strategies for Hadoop and Spark from HDFS on HPC clusters.

All these lead to the following broad challenges:

1) What criteria should we consider for efficient data access by Hadoop and Spark?
2) What alternative access strategies can be proposed for Big Data analytics on HPC clusters?
3) How can we re-design HDFS to accommodate the enhanced data access strategies?
4) Can these data access strategies also benefit different Big Data benchmarks and applications?

B. Contributions

In this paper, we consider a set of novel approaches to address the above-mentioned broad challenges.

1) We demonstrate that storage type is an important factor for efficient data access by Hadoop and Spark on HPC platforms with heterogeneous storage.
2) We propose enhanced data access strategies for Hadoop and Spark on HPC clusters with heterogeneous storage characteristic.
3) We re-design HDFS to accommodate the proposed access strategies so that Hadoop and Spark frameworks can exploit the benefits in a transparent manner.

Our evaluations show that in the presence of heterogeneous nodes with different types of storage devices, storage types (in addition to data locality) should also be considered while accessing data. Our proposed access strategies (e.g. Greedy (prioritizes storage type over locality), Hybrid (balances the load for locality and high performance storage), etc.)) can increase the read throughput of HDFS by up to 33%. Our proposed schemes also reduce the execution time of Hadoop and Spark Sort by up to 32% and 17%, respectively. The performances of Hadoop and Spark TeraSort are also improved by up to 11% by our design. To the best of our knowledge, this is the first study on incorporating storage types with data locality for accessing the HDFS data by Hadoop MapReduce and Spark applications on HPC clusters.

The following describes the organization of the paper. Some background knowledge is discussed in Section II. Section III describes our proposed design. The results of our evaluations are presented in Section IV. In Section V, we discuss the related work. We conclude the paper in Section VI with future works.

II. Background

In this section, we provide some background information related to this paper.

A. Hadoop MapReduce

Hadoop [30] is a popular open source implementation of the MapReduce [13] programming model. Two most important building blocks for a MapReduce program are the `map` and `reduce` functions provided by user. Although several execution frameworks have been proposed in the literature to handle MapReduce programs, the default execution model, implemented in Hadoop, is the most commonly used. In a typical Hadoop cluster, the resource management and the job scheduling functionalities are performed by a global Resource Manager. This Resource Manager is responsible for assigning resources to all the jobs running on the Hadoop cluster. The Node Manager is responsible for handling the tasks and the resources on each node with one Node Manager running per node. For each job/application, there is an Application Master. This process coordinates with the Resource Manager and the Node Managers to execute the corresponding tasks. It also orchestrates the launching, execution, and termination of several map and reduce tasks to maintain a successful completion of a MapReduce application.

B. Spark

While MapReduce [13] has revolutionized Big Data processing for data-intensive batch applications on commodity clusters, it has been demonstrated to be a poor fit for low-latency interactive applications and iterative computations, such as machine learning and graph algorithms. As a result, newer data-processing frameworks such as Apache Spark [34] have emerged. Spark is an in-memory, data-processing framework, originally developed in the AMPLab at UC Berkeley, that caches datasets in memory to reduce access latency. Its architecture is dependent on the concept of a Resilient Distributed Dataset or RDD [33], which is a fault tolerant collection of objects distributed across a set of nodes that can be operated on in parallel.

C. Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) is the primary distributed storage used by Hadoop and Spark applications. It is designed to store very large data sets in a reliable manner and provide user applications with high throughput access to data. An HDFS cluster follows a master-slave architecture, with one (or more) NameNode as the master that manages the file system namespace and regulates access to files. HDFS also has a number of DataNodes, usually one per node, acting as slaves that manage storage attached to their corresponding nodes. It supports a traditional hierarchical file organization and reliably stores very large files across machines in a large cluster by using replication for fault tolerance. By default, HDFS keeps three replicas for each block of data that it stores.

D. Default Data Access Strategy of Hadoop and Spark

HDFS is the underlying file system for Hadoop MapReduce and Spark. Both of these frameworks read data from HDFS while executing jobs. By default, the scheduler launches the tasks considering data locality in order to minimize data movements. The launched tasks read the data in HDFS without considering the storage type or bandwidth of the storage devices on the the local node.

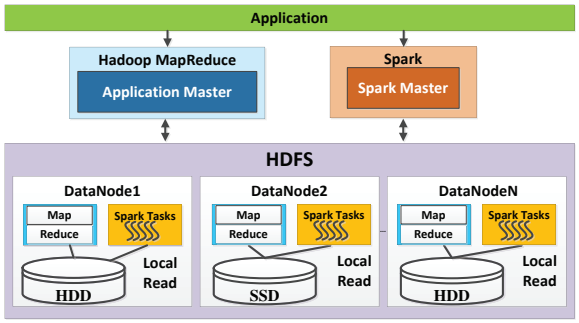


Figure 2: Default data access strategy

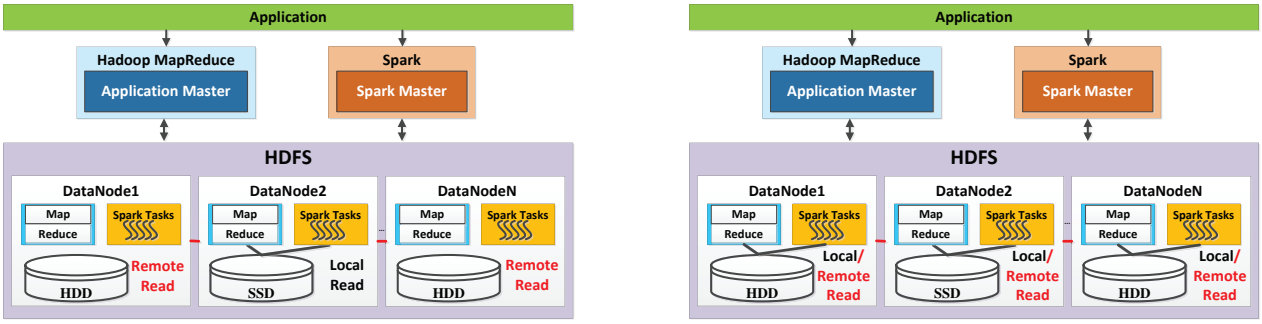
Figure 2 shows the default data access scheme by Hadoop and Spark in HDFS. In order to efficiently utilize the different types of storage devices in an HPC cluster with heterogeneous storage characteristics, a range of data placement policies have been proposed in the literature [14, 17]. These placement policies store different replicas of a data block to different types of storage devices considering performance, fault tolerance, and resource utilization. For example, the One_SSD policy stores one replica of a block in SSD and the others in HDD. In spite of this, Hadoop and Spark frameworks access data based on only data locality. As a result, tasks may read the data in HDD even though there is a copy of the same data in a high performance storage like SSD. Moreover, the presence of high performance interconnects in HPC platforms facilitates remote reads from faster storage devices. Therefore, both Hadoop and Spark need new data access strategies for HPC platforms.

III. Proposed Design

In this section, we propose different data access strategies for Hadoop and Spark on HPC clusters. We also present the associated designs introduced in HDFS in order to realize the enhanced read strategies.

A. Data Access Strategies

1) Greedy: In the *Greedy* strategy, the remote high performance storage devices are prioritized over local low performance storage. This strategy is particularly suitable for cases, in which the local storage devices on the Hadoop cluster are mostly hard disks. But some nodes in the cluster are equipped with large memory or SSD and replicas of the data blocks are also hosted by these nodes in SSD or in-memory storage. For such cases, we propose to read data



(a) Greedy (b) Hybrid

Figure 3: Proposed data access strategies

from the remote SSD or in-memory storage in a greedy manner. Figure 3(a) demonstrates the Greedy strategy of data access. As illustrated in the figure, the tasks launched in DataNode1 and DataNodeN, instead of reading data from the local storage devices (HDDs), access the data in SSD storage in DataNode2 over the interconnect. Tasks launched in DataNode2 read data locally as they see that the local storage has higher performance than those on most of the other nodes in the cluster. When the performance gaps among the storage devices are large and the underlying interconnect is fast, this method performs better for large data sizes.

2) Hybrid: In the *Hybrid* strategy, data is read from both local as well as remote storage devices. Figure 3(b) depicts the Hybrid read strategy. Considering the topology, the bandwidth of the underlying interconnect and available storage devices, we propose different variations of the *Hybrid* read strategy. These are as follows:

Balanced: The purpose of this data access strategy is to distribute the load among the nodes in the cluster. In this strategy, some of the tasks read data from the local storage devices (irrespective of the type of the storage) and the rest read from the remote high performance storage. The percentage of tasks to read from local vs. remote storage can be indicated by a configuration parameter.

Network Bandwidth-driven (NBD): In this scheme, tasks send read requests to remote nodes with high performance storage. The number of read requests that can be efficiently served by the remote node is bounded by the NIC bandwidth. Therefore, when the remote node reaches the saturation point of its network bandwidth, it sends a `Read_Fail` message to the client. The client then reads the data available in its local node (if any), irrespective of the type of storage. Otherwise (for rack-local tasks), the read request is re-sent to another node that hosts a replica of the desired data.

As depicted in Figure 4, when Client3 sends the `Read_Req` to the DataNode at time `t3`, it is already

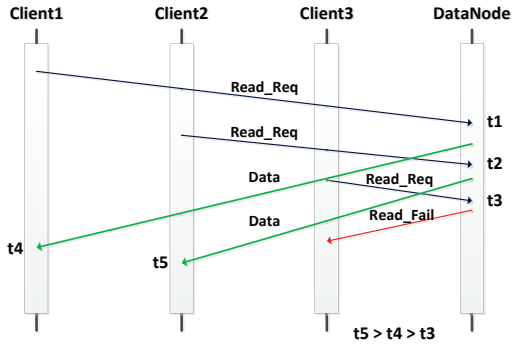


Figure 4: Overview of NBD Strategy

serving the requests from Client1 and Client2. The network bandwidth is saturated (assumption) at this point and therefore, the DataNode sends a `Read_Fail` message to Client3.

Storage Bandwidth-driven (SBD): In this scheme, tasks send read requests to remote nodes with high performance storage. The number of read requests that can be efficiently served by the remote node is bounded by the storage bandwidth of that node. Therefore, when the remote node reaches the saturation point of its storage bandwidth, it sends a `Read_Fail` message to the client. The client then reads the data available in its local node, irrespective of the type of storage or retries to read from another node.

Data Locality-driven (DLD): This strategy is a trade-off between the topological distances among nodes and the high bandwidth offered by the high performance storage devices like SSD and RAM Disk. In this scheme, the client reads data from the local storage devices as long as it is available. In this scenario, the client does not take into account the types of storage devices on the local node. If the data is not found locally, the client chooses to read the replica from a remote node that has high performance storage. If multiple replicas are hosted by different remote nodes having the

same type of storage device, the client chooses the node (with high performance storage) that is topologically the closest.

B. Design and Implementation

When Hadoop, Spark, or similar upper layer frameworks access data from HDFS, they send read requests for the required files to the HDFS client side. The client contacts the NameNode to get the list of blocks for the requested file. The client then chooses the best DataNode to read the blocks (each file is divides into multiple blocks) from. In order to incorporate the proposed enhanced read strategies, we introduce some new components in HDFS. Figure 5 shows the architecture of our proposed design.

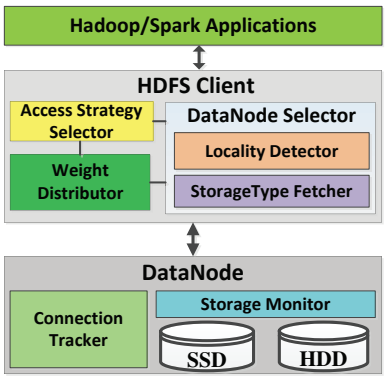


Figure 5: Proposed design

The new design components introduced in HDFS client side are:

Access Strategy Selector: HDFS can run with any of the proposed access strategies based on a user-provided configuration parameter. The client retrieves the selected strategy and passes it to the DataNode Selector.

DataNode Selector: The DataNode selector chooses the best DataNode to read a block based on the selected strategy and the available storage types in the cluster. It has two components:

1) *StorageType Fetcher*: The DataNode selector passes the list of DataNodes that hold the replicas of a block to the StorageType Fetcher. This module gets the storage types of the replicas from the NameNode. The NameNode has information of the storage types of all the replicas. The DataNode, after storing a block, sends the storage type to the NameNode in the block reports.
2) *Locality Detector*: The Locality Detector gets the list of nodes that host the replicas of the requested blocks and determines if the data is available locally or not. The list of DataNodes for a block is returned by the NameNode in a topologically sorted manner (since most of the blocks are replicated to three DataNodes by default, the length of this list is three) and, therefore, the first node in the list is the local node for data-local tasks.

Weight Distributor: This module is activated when the user selects the *Balanced* strategy. It gets the user-provide percentage and assigns weights (0 or 1) to the launched tasks accordingly based on their task ids so that the DataNode Selector can select some of the tasks (having weight 0) to read data locally and the rest (with weight 1) are redirected to the remote nodes with high performance storage devices.

In order to realize the proposed data access strategies, we modify the existing DataNode selection algorithm in HDFS by supplying the parameters obtained from **Locality Detector**, **StorageType Fetcher**, and **Weight Distributor**.

We choose the HDFS client side rather than the NameNode to host these components because if each client has the knowledge of the access strategy and gets the list of DataNodes (and storage types) that store the replicas of the blocks, each of them can compute the desired DataNode in parallel through the DataNode selection algorithm. Otherwise, the NameNode has to determine the desired DataNode for each client, which would be a bottleneck for large number of concurrent clients.

The design components introduced in the DataNode side are:

Connection Tracker: This component is enabled for the *NBD* scheme. It keeps track of the number of read requests N_c accepted by the DataNode. After N_c reaches a user-supplied threshold T_c (= Number of connections that saturate the network bandwidth), the DataNode does not accept any more read requests. When N_c becomes less than T_c due to some read requests being completed, the DataNode starts accepting read requests again.

Storage Monitor: This component is enabled for the *SBD* scheme. When the DataNode receives a read request, a reader thread is spawned to perform the I/O. This thread reads the data block from the storage device and sends it to the client. The **Storage Monitor** keeps track of the numbers of reader threads N_r spawned in the DataNode. After N_r reaches a user-supplied threshold T_r (= Number of concurrent readers that saturate the storage bandwidth), the DataNode starts sending Read_Fail message to the clients.

When *Greedy* or DLD scheme is selected by the user, the **StorageType Fetcher** and **Locality Detector** along with the enhanced DataNode selection algorithm can select the appropriate DataNodes to read the blocks from.

IV. Performance Evaluation

In this section, we present a detailed performance evaluation of Hadoop MapReduce and Spark workloads over HDFS using our proposed design. We compare the performances of different workloads using our enhanced data access strategies and compare them with those using the default read scheme. For our evaluations, we use Hadoop

2.6.0 and Spark 1.6.0. In all the experiments, we mention the number of DataNodes as the cluster size and set up half of the DataNodes with SSD storage while the rest have only HDD. For the *Balanced* strategy, we use a 50%-50% ratio of local vs. remote (high performance storage) reads.

In this study, we perform the following experiments: 1. Evaluations with TestDFSIO, 2. Evaluations with Hadoop Sort and TeraSort, and 3. Evaluations with Spark Sort and TeraSort.

For Hadoop Sort and TeraSort, we use the benchmark implementations that come with the Apache Hadoop distribution. For Spark workloads, we use the implementations provided by Intel HiBench [5].

A. Experimental Setup

We use three different clusters for our evaluations.

(1) OSU RI (Cluster A): This cluster has nine nodes. Each node has Xeon Dual quad-core processor operating at 2.67 GHz with 24 GB memory, 12 GB RAM Disk, two 1TB HDDs, and 300GB OCZ VeloDrive PCIe SSD. Each node is also equipped with MT26428 QDR ConnectX HCAs (32 Gbps data rate) with PCI-Ex Gen2 interfaces. This cluster also has 16 nodes each of which is equipped with 12GB RAM, 160GB HDD and MT26428 QDR ConnectX HCAs (32 Gbps data rate). The nodes run Red Hat Enterprise Linux Server release 6.1.

(2) OSU RI2 (Cluster B): We use 17 storage nodes in this cluster. These nodes are equipped with two fourteen Core Xeon E5-2680v4 2.4GHz. Each node is equipped with 512 GB RAM and one 2 TB HDD. Each node is also equipped with MT4115 EDR ConnectX HCA (100 Gbps data rate). The nodes are interconnected with two Mellanox EDR switches. Each node runs CentOS 7.

(3) SDSC Comet (Cluster C): Each compute node in this cluster has two twelve-core Intel Xeon E5-2680 v3 (Haswell) processors, 128GB DDR4 DRAM, 80GB HDD, and 320GB of local SATA-SSD with CentOS operating system. The network topology in this cluster is 56Gbps FDR InfiniBand with rack-level full bisection bandwidth and 4:1 over-subscription cross-rack bandwidth. Comet also has seven petabytes of Lustre installation.

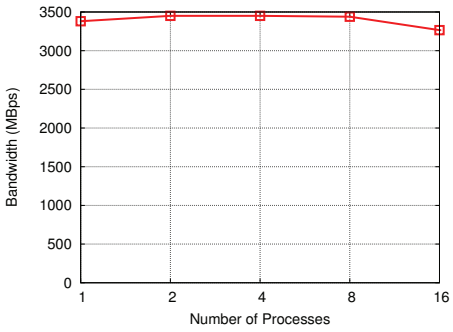


Figure 7: Selecting optimal number of connections in NBD strategy on Cluster A

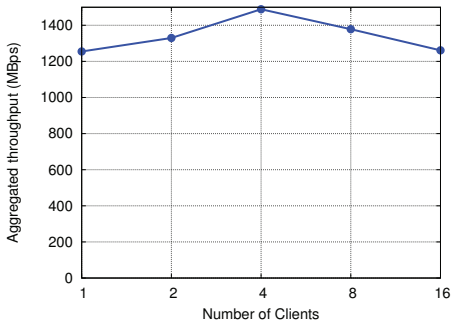


Figure 8: Selecting optimal number of readers in SBD strategy on Cluster A

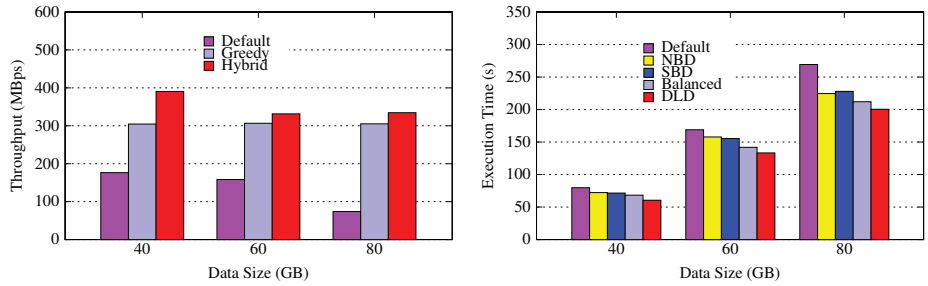
B. Evaluation with TestDFSIO

In this section, we evaluate the Hadoop TestDFSIO workload using our proposed data access strategies and compare the performances with those of the default (locality-aware read) scheme.

First we run our experiments on Cluster A on 16 nodes. Eight of these nodes have SSDs and the rest have only HDDs. Data placement is done following the One_SSD placement policy [14]. The experiments are run with eight concurrent maps per node. As observed from Figure 6(a), the Greedy scheme increases the read throughput by up to 4x over the default read scheme. The Hybrid read performs even better than the Greedy read by increasing read throughput by up to 11% over the Greedy read. The reason behind this is, when all the clients try to read greedily from the SSD nodes, at one point, the performance becomes bounded by the bandwidth of those nodes. On the other hand, the Hybrid scheme (DLD) distributes the load across all the nodes, making some clients read locally, while others perform remote reads from the SSD nodes.

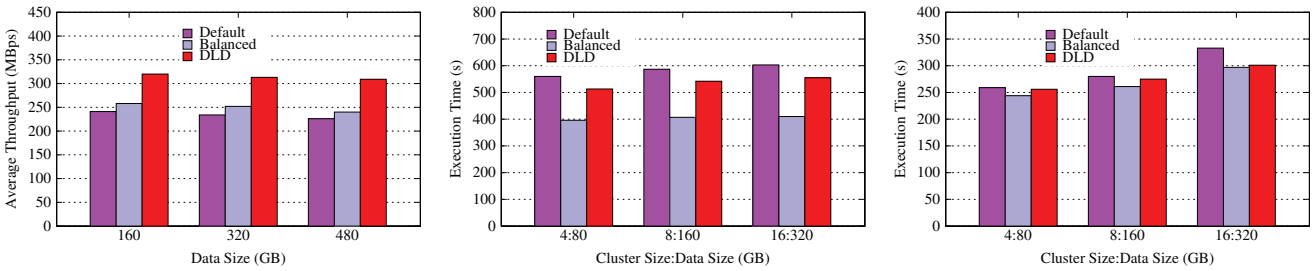
Next, we perform experiments to compare different Hybrid read strategies. These experiments are performed on eight nodes on Cluster A. Four of the nodes have SSDs and the rest have only HDDs. The experiments are run with eight concurrent clients per node. As observed from Figure 6(b), the Balanced scheme performs better than the default read in terms of TestDFSIO read latency. In the presence of many concurrent clients, distributing the load across multiple nodes in the Balanced scheme helps improve the performance. The DLD approach further improves the execution time by reading locally when available. But when remote reads are performed, the tasks select the nodes with SSD only, rather than reading from the remote HDD nodes. The number of rack-local tasks launched in this experiment is 22. NBD and SBD strategies initially work as Greedy but force the clients to read locally when network (NBD) or storage (SBD) bandwidth is saturated.

In order to determine the optimal number of connections supported by a DataNode in the NBD scheme, we run the OSU-MPI-Multi-Pair-Bandwidth (inter-node) [9] test on



(a) Evaluation of TestDFSIO Read (b) Comparison among the hybrid access strategies

Figure 6: Evaluation of the data access strategies on Cluster A



(a) TestDFSIO Read (b) Sort (c) TeraSort

Figure 9: Evaluation of Hadoop MapReduce workloads on Cluster B

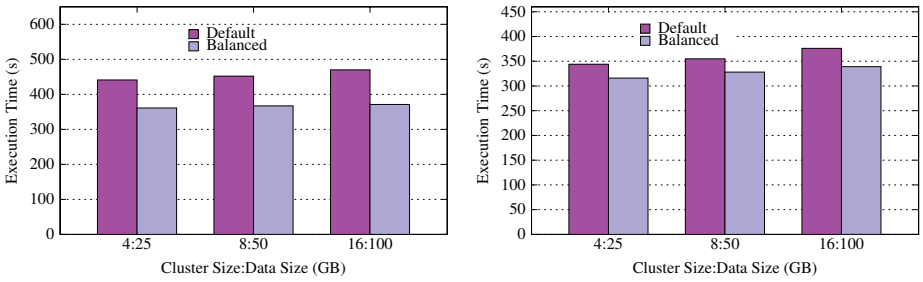
Cluster A with a message size of 512KB (HDFS packet size) and find that four connections provide optimal performance as depicted in Figure 7, as the bandwidth is maximized at this point. Similarly, to find out the optimal number of I/O threads that saturate the disk bandwidth, we run IOzone test on the SSD nodes with varying number of reader threads. Each thread reads a file (one file per thread) of size 128MB (HDFS block size) with a record size of 512KB. Figure 8 shows the results of our IOzone experiments. As observed from the figure, four concurrent readers maximize the total read throughput. Therefore, for the SBD approach, we use four reader threads as the threshold in the DataNode side. The reason that NBD and SBD approaches perform worse than Balanced is that, they force local reads under failure (local reads occur only when remote reads fail). But the Balanced approach does not have to go through the overhead of failed reads. For NBD and SBD, the loads on the DataNodes keep changing during the course of the job because when the requested blocks are sent to the client, the connection with that client is closed by the DataNode. Therefore, we send each read request (even after receiving a Read_Fail) from the HDFS client to the DataNode with high performance storage so that the client can read data from the remote storage once the DataNode is able to accept further requests.

We also evaluate the performance of TestDFSIO read on eight nodes on Cluster B. As observed from Figure 9(a), the read throughput is maximized by the DLD scheme. Both DLD and Balanced perform better than the default read approach in HDFS with DLD offering a benefit of up to 33%; DLD also outperforms Balanced by almost 20%. These tests are run with 24 concurrent tasks per node. In these tests, 20 rack-local tasks were launched. The figure clearly shows the benefits of the proposed access schemes. This further proves that in the presence of rack-local tasks (not data-local), it is always more efficient to read from the high performance storage devices through the proposed access schemes, rather than accessing data in the default approach.

C. Evaluation with Hadoop MapReduce Sort and TeraSort

In this section, we evaluate our design with the MapReduce Sort benchmark on Cluster B. For these experiments, the data size is varied from 80GB on 4 nodes to 320GB on 16 nodes. The experiments are run with 24 concurrent tasks per node. As observed from Figure 9(b), the Balanced read strategy demonstrates optimal performance for the Sort benchmark and reduces the execution time by up to 32% for 320GB data size on 16 nodes. Even though the DLD approach performs better than the default read strategy, Balanced works better than DLD. The reason behind this is that, the Balanced scheme distributes the I/O loads across multiple nodes while prioritizing the high performance storage nodes. Therefore, in the presence of concurrent tasks, it helps minimize the contention. Since Sort has equal amount of I/O, shuffle, and computation, minimizing the I/O pressure accelerates the other phases of the benchmark. Therefore, the Balanced approach performs better than DLD.

Figure 9(c) shows the results of our evaluations with Hadoop TeraSort. For these experiments, the data size is



(a) Sort (b) TeraSort

Figure 10: Evaluation of Hadoop MapReduce workloads on Cluster C

varied from 80GB on 4 nodes to 320GB on 16 nodes. The experiments are run with 24 concurrent tasks per node on Cluster B. The figure clearly shows that the Balanced read strategy reduces the execution time of TeraSort by up to 11% for 320GB data size on 16 nodes.

We further evaluate our design using Hadoop Sort and TeraSort benchmarks on Cluster C. On this cluster, we vary the data size from 25GB on 4 nodes to 100GB on 16 nodes. We use the Balanced strategy for data access here. As observed from Figure 10(a), our design reduces the execution time of the Sort benchmark by up to 21% for 100GB data size on 16 nodes. The benefit comes from the usage of enhanced data access strategy that accelerates the operations by balancing out the load of I/O. Similarly, our experiments with TeraSort on Cluster C yields an improvement of 10% over default HDFS. Figure 10(b) shows the results of our evaluations with TeraSort.

D. Evaluation with Spark Sort and TeraSort

In this section, we evaluate our design with the Spark Sort benchmark on Cluster B. For these experiments, the data size is varied from 80GB on 4 nodes to 320GB on 16 nodes. The experiments are run with 24 concurrent tasks per node. As observed from Figure 11(a), the Balanced read strategy demonstrates optimal performance for the Sort benchmark and reduces the execution time by up to 17% for 320GB data size on 16 nodes. Even though the DLD approach performs better than the default read strategy, Balanced works better than DLD.

Figure 11(b) shows the results of our evaluations with Hadoop TeraSort. For these experiments, the data size is varied from 80GB on 4 nodes to 320GB on 16 nodes. The experiments are run with 24 concurrent tasks per node on Cluster B. The figure clearly shows that the Balanced read strategy reduces the execution time of TeraSort by up to 9% for 320GB data size on 16 nodes.

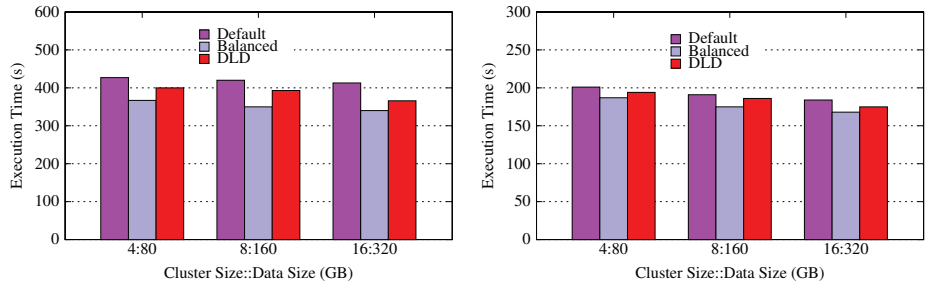
We further evaluate our design using the Spark Sort and TeraSort benchmarks on Cluster C. On this cluster, we vary the data size from 25GB on 4 nodes to 100GB on 16 nodes. We use the Balanced strategy for data access here. As observed from Figure 12(a), our design reduces the execution time of the Sort benchmark by up to 14% for 100GB data size on 16 nodes. Similarly, our experiments with Spark TeraSort on Cluster C yields an improvement of 9% over default HDFS. Figure 12(b) depicts the results of our evaluations with Spark TeraSort.

V. Related Work

Recently, improving the performance of data-intensive computing using heterogeneous storage support has been the focus of attention. In our first work along this direction, we proposed RDMA-enhanced designs [18] to improve HDFS performance, while keeping the default HDFS architecture intact. Later, we proposed a Staged Event-Driven Architecture [32] based approach to re-design HDFS architecture in [15]. Though we significantly reduced the architectural bottlenecks in HDFS by maximizing the possible overlapping during different stages, like communication, processing, and I/O, HDFS read-write operations still encounter noticeable I/O bottlenecks, as seen by many others [26, 27].

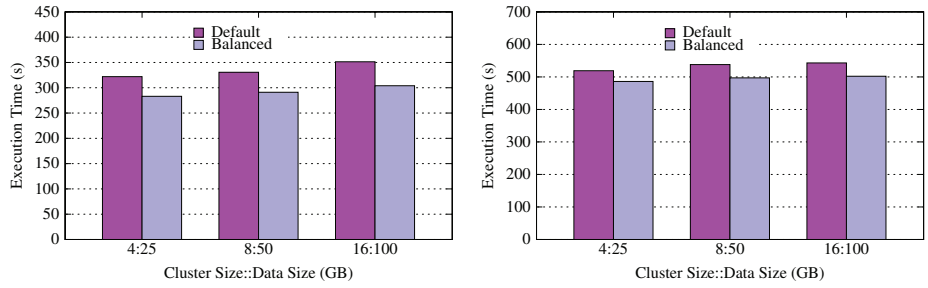
In this perspective, several research works [12] have shown that the HDFS read throughput can be improved by caching data in memory or using explicit caching systems. Similarly, the Apache Hadoop community has proposed explicit centralized cache management scheme in HDFS [29]. Along similar lines, Singh et. al presented a dynamic caching mechanism for Hadoop [28] by integrating HDFS with a Memcached-based metadata store for objects at HDFS DataNodes, to improve the HDFS read performance. In our recent works [16, 20], we have also studied on improving HDFS read and write performance by utilizing Memcached [7]. We cache the data blocks in Memcached to achieve huge performance benefits for HDFS. In parallel with these in-memory enhancements for improving MapReduce and other applications running over HDFS, new data processing frameworks, such as Spark [34], have emerged. Spark supports in-memory computing for large-scale Big Data processing.

With the rising use of HPC clusters for data-intensive computing, that are commonly equipped with heterogeneous storage devices such as, RAM, SSDs, and HDDs, several research works are being dedicated towards harnessing its power to minimize the I/O bottlenecks. In our recent work [17], we have introduced Triple-H, a novel design for



(a) Sort (b) TeraSort

Figure 11: Evaluation of Spark workloads on Cluster B



(a) Sort (b) TeraSort

Figure 12: Evaluation of Spark workloads on Cluster C

HDFS on HPC clusters with heterogeneous storage architecture. The use of hybrid storage usage patterns, RAM Disk and SSD-based Buffer-Cache, and effective data placement policies ensure efficient storage utilization and have been proven to significantly enhance the performance of Hadoop and Spark workloads [19]. Yet another in-memory file system, Tachyon/Alluxio [21], has shown promise in achieving high throughput writes and reads, without compromising fault tolerance via lineage. It also supports the usage of heterogeneous storage devices.

All of these above-mentioned file systems assume that each node on the cluster has a homogeneous storage architecture with RAM Disk, SSD, and HDD. But on many occasions, HPC clusters are equipped with nodes having heterogeneous storage characteristics, i.e. not all the nodes have the same type of storage devices. This paper addresses the challenges associated with obtaining high-performance in accessing data for Hadoop and Spark on such clusters.

VI. Conclusion

In this paper, we propose efficient data access strategies for Hadoop and Spark considering both data locality and storage types. For this, we re-design HDFS to accommodate the enhanced access strategies for reading data that are stored using placement policies that take heterogeneity into account. Our evaluations show that, the proposed data access strategies can improve the read performance of HDFS by up to 33% compared to the default locality-aware data access. The execution times of Hadoop and Spark Sort are also reduced by up to 32% and 17%. The performances of Hadoop and Spark TeraSort are also improved by up to 11% by our design.

In the future, we would like to evaluate the impact of the proposed data access strategies on other Big Data middleware and applications. We would also like to explore dynamic switching of the proposed access strategies.

References

[1] Apache Mesos. http://mesos.apache.org.
[2] Cloudera: Data access based on Locality and Storage Type. http://blog.cloudera.com/blog/2014/08/new-in-cdh-5-1-hdfs-read-caching/.
[3] Cloudera Spark Roadmap. https://2s7gjr373w3x22jf92z99mgm5w-wpengine.netdna-ssl.com/wp-content/uploads/2015/09.
[4] Future of Hadoop. https://www.datanami.com/2015/09/09/spark-is-the-future-of-hadoop-cloudera-says/.
[5] Intel HiBench. https://github.com/intel-hadoop/HiBench.
[6] King. https://king.com/.
[7] Memcached: High-Performance, Distributed Memory Object Caching System. http://memcached.org/.
[8] SuperCell. http://supercell.com/en/.
[9] The Ohio State Micro Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.
[10] Alex Woodie. Game-Changer: The Big Data Behind Social Gaming. https://www.datanami.com/2014/06/23/game-changer-big-data-behind-social-gaming/.
[11] Alex Woodie. What Pokemon GO Means for Big Data. https://www.datanami.com/2016/08/01/pokemon-go-means-big-data.

[12] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[13] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Operating Systems Design and Implementation (OSDI)*, 2004.

[14] Hadoop 2.6 Storage Policies. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/ArchivalStorage.html.

[15] N. S. Islam, X. Lu, M. W. Rahman, and D. K. Panda. SOR-HDFS: A SEDA-based Approach to Maximize Overlapping in RDMA-Enhanced HDFS. In *23rd International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2014.

[16] N. S. Islam, X. Lu, M. W. Rahman, R. Rajachandrasekar, and D. K. Panda. In-Memory I/O and Replication for HDFS with Memcached: Early Experiences. In *2014 IEEE International Conference on Big Data (IEEE BigData)*, 2014.

[17] N. S. Islam, X. Lu, M. W. Rahman, D. Shankar, and D. K. Panda. Triple-H: A Hybrid Approach to Accelerate HDFS on HPC Clusters with Heterogeneous Storage Architecture. In *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2015.

[18] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High Performance RDMA-based Design of HDFS over InfiniBand. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[19] N. S. Islam, M. W. Rahman, X. Lu, D. Shankar, and D. K. Panda. Performance Characterization and Acceleration of In-Memory File Systems for Hadoop and Spark Applications on HPC Clusters. In *2015 IEEE International Conference on Big Data (IEEE BigData)*, 2015.

[20] N. S. Islam, D. Shankar, X. Lu, M. W. Rahman, and D. K. Panda. Accelerating I/O Performance of Big Data Analytics on HPC Clusters through RDMA-based Key-Value Store. In *International Conference on Parallel Processing (ICPP)*, 2015.

[21] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *ACM Symposium on Cloud Computing (SoCC)*, 2014.

[22] X. Lu, N. S. Islam, M. W. Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda. High-Performance Design of Hadoop RPC with RDMA over InfiniBand. In *IEEE 42nd International Conference on Parallel Processing (ICPP)*, 2013.

[23] M. W. Rahman, N. Islam, X. Lu, and D. Panda. A Comprehensive Study of MapReduce over Lustre for Intermediate Data Placement and Shuffle Strategies on HPC Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 2016.

[24] M. W. Rahman, X. Lu, N. S. Islam, and D. K. Panda. HOMR: A Hybrid Approach to Exploit Maximum Overlapping in MapReduce over High Performance Interconnects. In *International Conference on Supercomputing (ICS)*, 2014.

[25] M. W. Rahman, X. Lu, N. S. Islam, R. Rajachadrasekar, and D. K. Panda. High-Performance Design of YARN MapReduce on Modern HPC Clusters with Lustre and RDMA. In *29th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2015.

[26] J. Shafer, S. Rixner, and A. L. Cox. The Hadoop Distributed Filesystem: Balancing Portability and Performance. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2010.

[27] K. Shvachko. HDFS Scalability: The Limits to Growth. *login: The Magazine of USENIX*, 2010.

[28] G. Singh, P. Chandra, and R. Tahir. A Dynamic Caching Mechanism for Hadoop using Memcached. https://wiki.engr.illinois.edu/download/attachments/197297260/\\ClouData-3rd.pdf?version=1\&modificationDate=1336549179000.

[29] The Apache Software Foundation. Centralized Cache Management in HDFS. http://hadoop.apache.org/docs/r2.3.0/hadoop-project-dist/hadoop-hdfs/CentralizedCacheManagement.html.

[30] The Apache Software Foundation. The Apache Hadoop Project. http://hadoop.apache.org/.

[31] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *ACM Symposium on Cloud Computing (SOCC)*, 2013.

[32] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[33] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[34] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2010.