hat*S*: A Heterogeneity-Aware Tiered Storage for Hadoop

Krish K.R., Ali Anwar, Ali R. Butt
Department of Computer Science
Virginia Tech
Blacksburg, VA
Email: {kris, ali, butta}@cs.vt.edu

***Abstract*—Hadoop has become the de-facto large-scale data processing framework for modern analytics applications. A major obstacle for sustaining high performance and scalability in Hadoop is managing the data growth while meeting the ever higher I/O demand. To this end, a promising trend in storage systems is to utilize hybrid and heterogeneous devices — Solid State Disks (SSD), ramdisks and Network Attached Storage (NAS), which can help achieve very high I/O rates at acceptable cost. However, the Hadoop Distributed File System (HDFS) that is unable to exploit such heterogeneous storage. This is because HDFS works on the assumption that the underlying devices are homogeneous storage blocks, disregarding their individual I/O characteristics, which leads to performance degradation. In this paper, we present hat*S*, a Heterogeneity-Aware Tiered Storage, which is a novel redesign of HDFS into a multi-tiered storage system that seamlessly integrates heterogeneous storage technologies into the Hadoop ecosystem. hat*S* also proposes data placement and retrieval policies, which improve the utilization of the storage devices based on their characteristics such as I/O throughput and capacity.**

We evaluate hat*S* using an actual implementation on a medium-sized cluster consisting of HDDs and two types of SSDs (i.e., SATA SSD and PCIe SSD). Experiments show that hat*S* achieves 32.6% higher read bandwidth, on average, than HDFS for the test Hadoop jobs (such as *Grep* and *TestDFSIO*) by directing 64% of the I/O accesses to the SSD tiers. We also evaluate our approach with trace-driven simulations using synthetic Facebook workloads, and show that compared to the standard setup, hat*S* improves the average I/O rate by 36%, which results in 26% improvement in the job completion time.

***Keywords*-Tiered storage; Hadoop Distributed File System (HDFS); data placement and retrieval policy.**

I. Introduction

In recent years, the implementation of MapReduce [10] in Hadoop [2] has emerged as an efficient framework that is being extensively deployed to support a variety of big-data applications [3], [13], [22]. The main challenge faced by researchers and IT practitioners in sustaining Hadoop is how to evolve the underlying storage and I/O infrastructure to deal with the exponentially growing data volumes, and to do so in an economically viable fashion.

A promising trend in storage technologies is the emergence of heterogeneous and hybrid storage systems [38], [26], [5], [4] that employ different types of storage devices, e.g., SSDs, ramdisks, etc. Moreover, the networking infrastructure bandwidth is growing at a pace that is an order of magnitude higher than the I/O bandwidth improvements in hard disk drives (HDDs) [9]. The two trends are enabling realization of distributed, hierarchical, hybrid and heterogeneous storage solutions that are efficient and cost effective, e.g., Hystor [5] and ConquestFS [35]. These systems typically integrate HDDs with fast emerging storage mediums, e.g., ramdisks, SSDs, etc. The faster storage serves as a buffer for frequently accessed data and yields very high I/O rates, while the HDDs store the infrequently accessed data and provide cheap high-capacity storage for the large data volumes.

TABLE I
Specifications of different storage devices used in the HDFS test, and the fraction of total Reads (*Grep*) and Writes (*TeraGen*) serviced by each.

Device Type	Write BW MB/s	Read BW MB/s	IOPS	# of devices	% of writes	% of Reads
PCIe SSD	245	533	70k	3	7	6
SATA SSD	139	191	25k	9	16	27
HDD	46	61	3.5k	27	77	67

Inspite of the above developments, it is a challenge to leverage advanced storage solutions in the context of Hadoop. This is because the Hadoop Distributed File System (HDFS) [32] — that serves as the storage substrate for Hadoop clusters — is not designed to handle heterogeneous storage. HDFS treats all the underlying storage components to be comprised of blocks with same I/O characteristics. Thus data is distributed uniformly across all the storage devices, irrespective of their I/O characteristics and capacity, which leads to inefficiencies and resource wastage. To highlight these problems, we ran two representative Hadoop applications, *TeraGen* and *Grep* [20], on a 28-node cluster. The cluster storage is provided by two types of SSDs (PCIe SSD and SATA SSD) and one type of HDD. The PCIe and the SATA SSDs have a measured IOPS rate that is 20× (9× random read bandwidth, 5× random write bandwidth) and 7× (3×, 3×) that of the HDDs, respectively. Thus, the SSDs are provisioned to serve more requests at a higher throughput than HDDs. Table I records the percentage of reads and writes for each device under the test runs. We observe that even though the faster SSDs were available, the device characteristics oblivious uniform data distribution of HDFS leads to a large amount of data stored on the HDDs. Thus the HDDs serviced 77% and 67% of the accesses for

TeraGen and *Grep*, respectively. This entails a performance loss of 72% and 61%, respectively, compared to the ideal case for the tests where all of the accesses were serviced by the SSDs that were available for use. This small test shows that HDFS is unable to exploit the benefits of individual devices in a hybrid setting.

One way to incorporate emerging storage devices into Hadoop is to equip the nodes with one type of device only, e.g. SSD of the same type. However, this is impractical as the cost per GB of such devices is still far from the economical storage offered by HDDs, and this cost gap is expected to remain high in the near future [23]. Thus cluster deployments are likely to adapt the hybrid approach of using HDDs along with a variety of storage devices. Moreover, large clusters typically go trough several upgrade phases [12] during their lifetime, thus all the nodes can not be expected to have homogeneous storage performance even if only HDDs are utilized. Yet another source of heterogeneity is the emergence of enterprise consolidated storage solutions for Hadoop [29], [30], [11], [20], which couple node-local storage with network-attached central storage to provide ease of data management while sustaining high I/O rates. Thus there is a need for enhancing the Hadoop storage layer to manage heterogeneity in the underlying storage systems.

In this paper, we explore the utility of heterogeneous storage devices in Hadoop and address challenges therein by designing hat*S*, a heterogeneity-aware tiered storage for Hadoop. hat*S* logically groups all storage devices of the same type across the nodes into an associated "tier." A deployment has as many tiers as the different type of storage devices used, and a node with multiple types of devices is part of multiple tiers. For instance, if a deployment consists of nodes with a SSD and a HDD, all the SSDs across the nodes will become part of a SSD tier, and similarly all the HDDs will form the HDD tier. By managing the tiers individually, hat*S* is able to capture the heterogeneity and exploit it to achieve high I/O performance.

Contrary to HDFS that only considers network-aware data placement and retrieval policies, hat*S* proposes additional policies to replicate data across tiers in a heterogeneity-aware fashion. This enhances the utilization of the high-performance storage devices by efficiently forwarding a greater number of I/O requests to the faster tier, thus improving overall I/O performances. To facilitate this, in addition to the standard HDFS APIs, hat*S* also provides custom APIs for seamless data transfer across the tiers and management of stored data in each tier. These features allow hat*S* to integrate heterogeneous storage devices into Hadoop to extract high I/O performance. Specifically, we make the following contributions in this paper:

- Design and implement hat*S*, a novel enhancement for HDFS, which considers storage characteristics as a part of the property of the associated node, and provides the ability to distinguish between different types of storage devices attached to Hadoop DataNodes. This ability can then be used to match application I/O needs with appropriate storage devices.
- Implement data placement and retrieval policies based on storage device characteristics, which enables utilization of high speed storage devices in an efficient manner.
- Provide APIs to easily specify application storage needs, and to move replicas between tiers automatically based on characteristics such as performance and cost, so as to extract high I/O performance from the storage system.
- Validate hat*S* design and techniques therein using in-depth simulations and experiments on a real deployment.

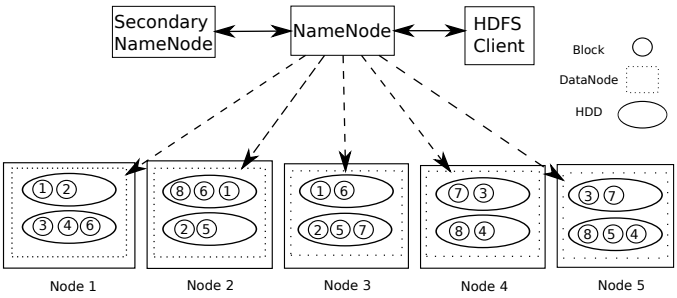


Fig. 1. HDFS architecture with default data placement policy.

We evaluated hat*S* using a deployment on a 28-node Hadoop cluster equipped with three types of storage devices (Table I). For each of the studied Hadoop applications, namely *TeraGen*, *Grep*, and *TestDFSIO*, hat*S* improved the utilization of the PCIe SSDs by 91% by servicing 64% more requests from the SSDs than HDDs. Our analysis of data placement and retrieval policies on these applications shows that hat*S* stores 37% more and retrieves 33% more data from the SSDs, compared to the default HDFS policies. This improves the read throughput of the storage system by 32.6%. We replay a publicly available synthetic Facebook production trace in our simulation framework and observe that hat*S* improves the average I/O rate of HDFS by 36% and the overall execution time for the trace by 26%.

II. Background: Hadoop Distributed File System

A node in a Hadoop deployment consists of both compute processors and directly-attached storage. A small number of nodes (typically $12 - 24$ [12]) are grouped together and connected with one network switch to form a rack. A Hadoop cluster may consist of one or more such racks. Figure 1 shows the architecture of HDFS that provides data management and storage. The main functions of HDFS are to ensure that tasks are provided with the needed data, and to protect against data loss due to failures. HDFS consists of a master data management component, *NameNode*, which manages worker components called *DataNodes* running on each node. All the nodes also run corresponding components for task execution, i.e., *JobTracker* and *TaskTrackers* [37].

HDFS divides all stored files into fixed-size blocks (chunks) and distributes them across DataNodes. Moreover, the system typically maintains three replicas of each data block, two placed within the same rack and one on a different rack. Unlike traditional high-performance computing clusters, all storage needs are supported by HDFS using node-attached storage, and no network-attached consolidated storage is necessary (though may be used for additional backups etc.). Thus,

there are two reasons for using replication in HDFS. First, Hadoop deployments often consist of less reliable off-the-shelf commodity machines or are at such a scale that failure is the norm and not an exception [28]. Replication prevents data loss due to such node failures. Second, having multiple copies of a block means that they can be read in parallel, thus providing higher I/O rates. Replication also improves the chance of finding a block with better proximity to the node on which the application is running, thus reducing the network cost of sending the request across racks [10], [32], [37].

The extant implementation of HDFS considers all storage devices managed by DataNodes to be homogeneous, irrespective of the I/O characteristics of the devices [10]. The storage and retrieval schemes supported by HDFS are optimized from the perspective of network utilization. When storing a block, the HDFS client component—that handles the I/O requests for the application—queries the NameNode for a set of potential DataNodes on which to store the multiple replicas of the block [32], [37]. The NameNode normally replies with: the local machine on which the application instance storing the block is running, a randomly chosen node on the same rack that contains the local machine, and a randomly chosen node from a rack different from the local machine's. This policy provides resilience against both node and rack failures. However, storage availability and other constraints may change the replica placement, e.g., if a local machine does not have enough storage space a local replica cannot be generated. Similarly, for retrieving a file, the client queries the NameNode for nodes that store a replica of the file blocks, and receives a DataNode list sorted according to the network proximity to the client. The client uses the list along with the load on the DataNodes to select nodes from where to retrieve the blocks.

As highlighted in the example of Table I, HDFS does not consider storage characteristics in data placement and retrieval. Thus, if heterogeneous storage devices are employed, the current implementation will not be able to exploit the full performance potential of the devices. The new implementation of HDFS in Hadoop 2.0 [34] continues to face the same challenges from storage heterogeneity. This is because the focus in Hadoop 2.0 is to remove central points of failure by supporting multiple NameNodes and allow for a larger namespace than was previously possible. However, all storage on a node is still managed by a single DataNode that has the implicit homogeneity assumption. Thus, innovation is needed to enhance and adapt HDFS for emerging storage trends.

III. Design

hat*S* enhances HDFS for heterogeneous storage devices by creating a storage hierarchy based on the performance characteristics of the devices, and designing heterogeneity-aware data placement and retrieval policies that improve overall I/O performance in Hadoop clusters.

A. System Architecture

Figure 2 shows the overall architecture of hat*S*. An important difference between hat*S* and HDFS is the design of the

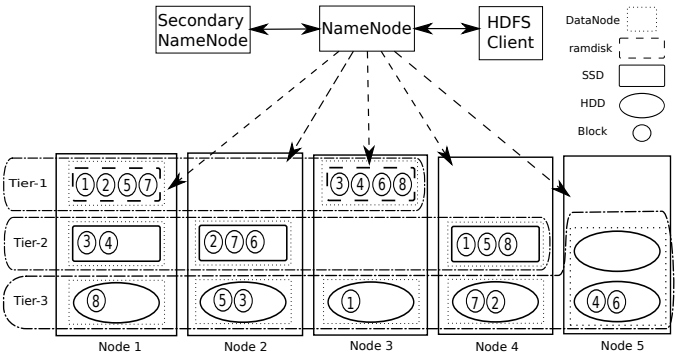


Fig. 2. hat*S* architecture overview.

DataNode. In HDFS, each participating node hosts a single DataNode instance, constituting multiple storage devices, regardless of their characteristics such as supported I/O rates and capacity. In contrast, each participating node in hat*S* hosts multiple DataNode instances, where each instance represents only one type of storage device. For example, a node with two HDDs and a SSD will have two DataNodes in hat*S*, one associated with the HDDs and the other with the SSD.

All devices of the same type and similar I/O characteristics, e.g., all similar HDDs, across all the participating nodes are logically grouped into a virtual storage "tier." For example, a tier of HDD Type X will encompass all DataNode instances in a deployment that are associated with Type X HDDs attached to the nodes. This enables hat*S* to not only capture the unique characteristics of heterogeneous storage devices but also distinguish between different storage tiers and utilize them accordingly. To achieve this, we modify the DataNode to also include a tier identifier as part of its characteristics specification. At the time of cluster configuration the administrator specifies the tiers for the DataNodes. We also modify the NameNode to use tier identifiers to group the DataNodes into their associated tiers. Each node can be part of multiple tiers depending on the devices that are attached to it. Moreover, a tier typically will have only one kind of device; but multiple kinds, such as HDDs that have only slightly different I/O characteristics, can also be associated with the same tier at the administrator's discretion. The number of DataNodes making up a tier vary based on the hardware composition of the cluster. In the example shown in Figure 2, Node 1 has three DataNodes belonging to three different tiers and Node 5 has only one DataNode belonging to Tier-3. Tier-1, Tier-2 and Tier-3 have 2, 3 and 5 DataNodes, respectively.

hat*S* exploits the tier information to strategize when and where to place replicas of a block. We discuss several data placement policies in the subsequent section, however, hat*S* maintains the invariant that a tier contains all blocks belonging to a file. This is to avoid dividing a file across a slow and a fast tier, where the slow tier devices will become the bottleneck and negate the benefits of the fast devices. Moreover, a tier can have more than one replica of a file, and a file can be replicated in multiple tiers as long as each tier contains a complete copy of the file. This provides for routing accesses to frequently used files to faster tiers, and relegating the infrequently used

files to slower tiers. Note that, this approach still provides for replicating data within and across racks as in standard HDFS, but imposes the additional constraint of keeping a complete copy of a file in a given tier. One concern is that the whole copy invariant may be violated in case of node failures. To overcome this, we introduce a new monitoring daemon on the NameNode, which ensures that any re-replication is done in a tier-aware fashion.

B. *hat*S *Data Management APIs*

In addition to the file system APIs provided by HDFS, hat*S* provides new APIs specified in Table II to manage the tiered storage. The main functions of hat*S* include associating DataNodes to appropriate administrator-specified tiers, and providing data access based on different policies. DataNodes are added to the tier during initialization only, and can not be modified at runtime. This is because the I/O and capacity characteristics that are considered in our tiers are specific to the devices used and do not change while the system is running. Thus, the main runtime APIs allow the system to move files between tiers, create a replica of an already existing file in a specified tier, delete a file from a specified tier, and create new replicas based on specified replica management policy. We note that, similarly as in HDFS, all the APIs modify data placement in the granularity of a file and do not support block level modification.

C. Data Placement and Retrieval Policies

A challenge in hat*S* is to determine when and where the data, i.e., a replica of a file, should be placed. This is a crucial design decision as naive replication can compromise performance and reduce the efficacy of our approach. Moreover, since hat*S* like HDFS is a write-once read-many file system, provisioning for efficient data retrieval is also crucial for improving overall system I/O performance, and depends on the placement policy employed. In the following, we describe several data placement and retrieval policies that we consider.

1) Network-Aware Policy: The first policy that we consider is the default network-aware placement and retrieval used in HDFS (Section II). The placement policy distributes each data block across multiple racks to ensure fault tolerance against node and rack failure. However, network-aware data placement does not take into account the performance of underlying storage devices. Under this policy, the blocks are randomly distributed across DataNodes in a rack, so a file may be replicated across tiers such that the portion of the file stored in a tier will depend on the number of DataNodes in that tier. Similarly, for data retrieval a list of DataNodes ordered with respect to network proximity to the application instance is obtained from the NameNode and the nearest replicas are used. While this approach reduces network traffic, the nearest replica can be on a slower device. In contrast, a more distant but faster replica could have provided higher overall I/O and would have been a better alternative. Network-aware retrieval is oblivious to this information and hence cannot leverage such heterogeneity-based trade-offs. Thus, this policy is not a good match for hat*S* as it crosses tier boundaries and will lead to performance imbalance when multiple types of devices are involved, e.g., SSDs and HDDs. Moreover, given that not all tiers are expected to be provisioned with the same amount of storage, and there will be more DataNodes in HDD tier (given the low $/GB) than tiers containing expensive devices such as SSDs and ramdisks, this policy will direct most accesses to the slower devices even when faster devices are available in the system.

2) Tier-Aware Policy: The next policy we consider is tier-aware placement and retrieval, which takes into account the storage characteristics of the underlying storage devices and completely replicates a file in multiple tiers. For clusters having more than one storage tier, we replicate the file to up to three different tiers. For clusters with more than three tiers, we chose the first replica to be placed in a fast tier, the second in a slow tier, and the third in a randomly chosen tier with intermediate performance. Since the first replica is treated as a source for the second replica [32], [37], storing the first replica on a faster tier will also speedup the replication process. Tier-aware placement does not consider the underlying network infrastructure as such, and only ensures that a node stores a single replica even if the node has multiple DataNodes. This prevents data loss in case of node failure, as the replica can be re-created from other sources.

For data retrieval, the ideal would be to always access the data from the fastest tier. However, doing so in Hadoop will result in hotspots where some DataNodes are overloaded, and will affect the performance of the system. Moreover, given that the capacity and number of faster tiers is limited, retrieving data only from the fastest available tier will also entail higher cross-rack network traffic and related overheads. To this end, we associate a weight to each tier from which a block can be retrieved, and then employ a weighted random function to determine which DataNode to use for retrieval. The assigned weights of each tier are determined using storage characteristics, such as IOPS and capacity, of the DataNode. This approach is effective in distributing the requests to a file among multiple tiers and each tier will serve varying number of blocks. For example, an SSD with $70k$ IOPS will be able to serve at least $10\times$ more request than a HDD with $3.5k$ IOPS as we show later in the evaluation (Section V).

While this policy takes into account tier characteristics, all the replicas of a block may be stored on one rack, and the data may be exposed to rack failures. Moreover, replication of blocks across racks is desirable for load balancing and providing better data locality for read operations. To avoid such data placement skewness, network characteristics have to be considered along with tier information, which we do in our next approach.

3) Hybrid Policy: Tier-aware data placement and retrieval policy improves the I/O performance by making replicas in specific tiers, while network-aware placement improves resilience by making replicas across racks. For improving I/O throughput, reducing cross-rack traffic to efficiently use the network, and fault tolerance, we need to have replicas

TABLE II
HAT*S* APIS TO ENHANCE HDFS.

API	Arguments & Return Type	Description
boolean createFileTier(...)		**Creates a replica of a file in the specified tier.**
	String `filename`	Name of the file to be replicated.
	String `tier`	Tier in which the replica will be created.
	boolean `return_value`	Returns 0 on success, 1 on failure.
boolean deleteFileTier(...)		**Removes a replica of a file from a tier.**
	String `filename`	Name of the file whose replica will be deleted.
	String `tier`	Tier from which to remove the replica.
	boolean `return_value`	Returns 0 on success, 1 on failure.
boolean moveTier(...)		**Moves replicas of a file across tiers.**
	String `filename`	Name of the file to be moved.
	String `from_tier`	Source tier from which replica will be removed.
	String `to_tier`	Destination tier for the new replica.
	int `number_of_replicas`	Number of file replicas to be moved.
	boolean `return_value`	Returns 0 on success, 1 on failure.
Void setRepPolicy(...)		**Modifies the replication policy for a file.**
	String `filename`	Name of the file to be affected.
	String `policy`	Storage policy to use.
	int `number_of_replicas`	Number of replicas under the new policy.

across tiers as well as across racks. To this end, we design a hybrid network- and tier-aware data placement policy that works as follows. The first replica is placed with one of the DataNodes on the local node. The second replica is placed in a different rack than the one used for the first replica and also on a different storage tier than that of the first replica. The third replica is placed on a tier different from the other two replicas, but rack-local to either of the replicas. Moreover, the tier selection is done similarly as in the tier-aware placement policy. The key advantage of this policy is that it achieves the same replica distribution as that of standard HDFS, which is effective in ensuring high I/O with good resilience to failures, while also considering the heterogeneous storage characteristics.

Similarly, for retrieval we adjust the weights used in our tier-aware policy to also factor in network proximity and the cost of transferring a block over the network to achieve higher I/O throughput as well as to reduce cross-rack traffic.

While the hybrid policy has the same expected fault tolerance as in HDFS before a failure occurs, after a failure occurs special steps have to be taken by hat*S* in replica regeneration to ensure that a new replica is stored on an appropriate tier in addition to being on an appropriate rack. Moreover, if a DataNode is overloaded with requests or low on capacity, replica creation or regeneration may not be possible on appropriate tiers. However, we then utilize the monitoring daemon to detect placement anomalies and move the data to appropriate tiers.

D. Discussion

In this section, we discuss the impact of hat*S* on other cluster components. First, the Hadoop job scheduler is network-aware and aims to schedule jobs on or near nodes that hold the needed data. Our hybrid approach preserves the network proximity, thus no change is required in the scheduler to avail the higher I/O rates offered by hat*S*.

Second, hat*S* tries to utilize faster tier resources whenever possible. However, the number of such devices is likely to be limited given their high cost. This would mean that the faster tiers may quickly become full, and the applications needing more data can no longer benefit from them. We remedy this by using the monitoring daemon along with replica movement APIs to flush the unused data from the faster to slower tiers.

Third, hat*S* requires nodes to run multiple DataNode instances instead of just one as in standard HDFS. This can potentially increase the load on the node and affect performance. We argue that this additional overhead is distributed across all the nodes and is negligible because of the following reasons. (i) The different types of devices attached to a node is expected to be small. (ii) The total number of blocks stored on the node is similar as under HDFS and is independent of the number of DataNodes. The in-memory data structures at the NameNode depend on the number of data items and number of replicas, but not on the number of DataNodes. Since, we do not increase the number of replicas, we expect this factor to be the same as well, so hat*S* is not expected to add any significant overhead to the NameNode. (iii) The overhead associated with accessing a block is also similar to HDFS, as hat*S* modifies only the metadata space of these blocks.

Fourth, hat*S* proposes to utilize SSDs in the Hadoop storage tier. There is a concern that such devices have limited erase cycles, and may affect the MTTF. We argue that incorporating SSDs in Hadoop is not unique to our approach, and other state-of-the-art works have also purported the same. Moreover, numerous SSD optimization approaches are available [33], [7] to remedy this. Thus, SSD endurance is orthogonal to our design; is useful even in when no SSDs are used but different kinds of HDDs are employed.

In summary, hat*S* provides a variant of HDFS, which considers the characteristics of the underlying storage devices and network infrastructure for its data access policies, thus yielding improved I/O performance.

IV. IMPLEMENTATION

We have implemented hat*S* as described in Section III. In total, we modified or added about 1800 lines of Java code in Hadoop 0.20.1 to add the features of tiering and heterogeneity awareness and to enable the APIs of Table II.

a) Tier identification: We modify the `hadoop-daemon.sh` script to enable a Hadoop node to have multiple logical DataNodes, and to coalesce DataNodes with similar storage characteristics into respective tiers. We introduce a new parameter *dfs.tier.id* in the Hadoop configuration file (`hdfs-site.xml`), which the cluster administrator can use to identify the tiers for the different storage devices. Next, we modify HDFS's *DataNodeDescriptor* data structure to incorporate the tier information as an additional global characteristic of each DataNode. The extended descriptor can then be used by the HDFS's *DataNodeRegistration* process for registering the tier-based DataNode with the NameNode.

b) Data placement: To support data placement policies based on storage device characteristics, we modify the NameNode's *ReplicationTargetChooser* component to implement different data placement schemes. A list of nodes is chosen from the *NetworkTopology* structure that provides information about various racks and tiers in the cluster (*clusterMap*). To ensure that a DataNode is not used to store multiple replicas of the same block, we re-purpose the block-specific *excludenode* list by adding the already chosen DataNode as well as the other DataNodes on the same node to the list. This results in a node having only one copy of a block as desired.

After a DataNode is chosen to store a block, the *block* and its corresponding *INodeFile* structure are associated with the DataNode's tier. This is to enable re-replication of the block in the same tier in case of a failure. A background daemon periodically runs to ensure that the blocks are associated with the appropriate tier, and if not, the daemon initiates our *moveTier* API to move the replicas to the appropriate tiers.

c) Data retrieval: Data retrieval in HDFS uses weighted random approach to select a replica from the list of DataNodes that store the data. To support our different retrieval policies, we implemented weighted random methods in *NetworkTopology*. The weights can be re-adjusted for this selection based on the policy, i.e., network-aware, tier-aware or hybrid. For instance, network-aware scheme will assign weights to a DataNode based on its proximity to the client, tier-aware scheme will assign weights based on the characteristics of the storage device that the DataNode supports, and hybrid scheme will consider both the factors. In our current implementation, we have used fixed hard-wired values for the weights as our testbed characteristics are known to us a-priori, but in a real setup, the administrator can specify the weights in a configuration file.

V. Evaluation

In this section, we present the evaluation of hat*S* using both a real deployment on a medium-scale cluster and simulations. We compare the effectiveness of different data placement and retrieval policies discussed in Section III-C, and their impact on the I/O performance of Hadoop jobs. For comparison, we also consider a random data placement and retrieval policy, which is oblivious of both tier and network information.

A. Experimental Setup

Our testbed consists of a master node and 27 worker nodes configured in three racks of nine nodes each. The nodes have two 2.8 GHz quad-core Intel Xeon processors, 8 GB of RAM, and one SATA HDD. The HDDs are 500 GB 7200 RPM Seagate Barracuda ES.2 drives. In addition to HDDs, three of the worker nodes in each rack are provisioned with an Intel 520 series 128 GB SATA SSD and one worker node in each rack is provisioned with an additional OCZ RevoDrive series PCIe 128 GB SSD. Table I shows the performance specifications of these storage devices. In our setup, Tier-1, Tier-2, and Tier-3 contain all the DataNodes that are equipped with the PCIe SSDs, the SATA SSDs, and the HDDs, respectively. Moreover, a node is associated with at most two tiers. The nodes are connected using both a dedicated 1 Gbps Ethernet switch as well as a dedicated 10 Gbps InfiniBand switch. We use InfiniBand as our default interconnect, using the slower connection only where specified in the following discussion. Each worker node is configured with six map slots and two reduce slots so as to use all of the available cores on the node. The considered benchmarks are mostly map intensive, so there are more map slots than reduce slots.

The master node runs both the Hadoop JobTracker and NameNode for all the experiments, and all the worker nodes contribute to both TaskTracker and DataNode. Worker nodes with more than one type of storage devices have multiple DataNodes, thus our testbed has 39 DataNodes co-existing with 27 TaskTrackers. As the focus of our experiments is to study the impact of HDFS I/O operations, the intermediate shuffle data is stored on the HDDs local to the TaskTracker. The replication factor is fixed at the default three, and the block size used is 64 MB.

B. Performance Under Different Policies

We analyze the read and write performance of hat*S* under different data placement and retrieval policies using the HDFS benchmark *TestDFSIO*. Each worker node writes a 1024 MB file (16 blocks) during the write test and reads a file of the same size during the read test. Figure 3 shows the results for the write accesses. We measure the overall I/O throughput for each of the map tasks and calculate the average I/O rate across all map tasks. We observe that the network-aware and hybrid policies behave similarly. This is because the write operation succeeds after it writes to the OS buffer cache and does not wait for the data to be synced to the storage device. We see a reduction in the throughput and average I/O rate for the tier-aware and random policies, which is expected as they do not consider network proximity and associated overhead.

Figure 4 shows the *TestDFSIO* results for the read test. As the hybrid policy considers network proximity and tier information, it offers significantly higher I/O rates than the other studied policies. Similarly as in the write test, the network-aware and the tier-aware policies perform better than the random policy. An interesting observation here is that the network-aware policy has a higher throughput than the tier-aware policy, whereas the average I/O rate of the tier-aware

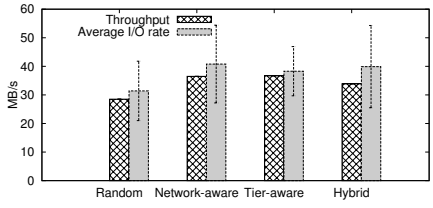


Fig. 3. Overall write throughput and average I/O rate per map task in *TestDFSIO-Write* under the studied policies.

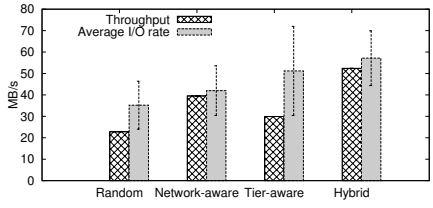


Fig. 4. Overall read throughput and average I/O rate per map task in *TestDFSIO-Read* under the studied policies.

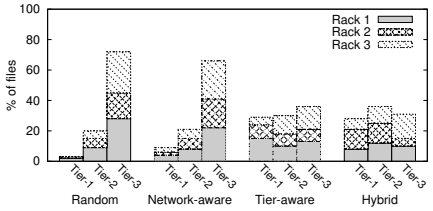


Fig. 5. Data distribution in terms of tiers and racks under the studied data placement policies, normalized to the total data stored for each run.

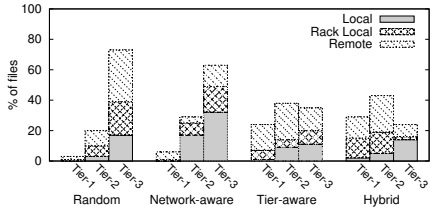


Fig. 6. The breakdown of reads based on tier and network proximity. The y-axis is normalized to the total read accesses in each run.

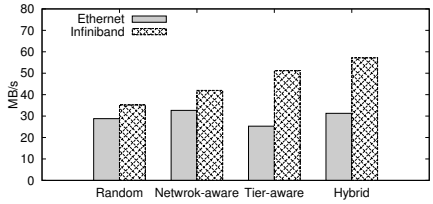


Fig. 7. Average I/O rate observed for InfiniBand and 1 Gbps Ethernet connections under studied data management policies.

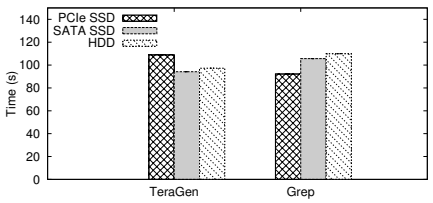


Fig. 8. Comparison of execution time observed for the different storage devices.

policy is better than that of the network-aware policy. The tier-aware policy is network oblivious, thus the probability of using a local fast tier for an access is similar to that of using a remote slow tier, which is seen as a high standard deviation in the average I/O rates in the Figure. Moreover, the tier-aware policy aggressively tries to utilize the storage devices in the fast tier without considering the network constraints. This sometimes results in network contention, causing the I/O rate to be quite low for some map tasks. We also observe that the hybrid policy offers 32.6% better I/O throughput and 36% better average I/O rate compared to that of the default network-aware policy.

C. Impact of Placement Policy

In our next experiment, we used *TeraGen* to generate a 27 GB file consisting of 432 blocks. By default, *TeraGen* uses only two mappers to generate all of its data, so each TaskTracker generated 13.5 GB. In the network-aware and hybrid placements, while storing the local replica of the data, all the blocks of the 13.5 GB file will be skewed to one DataNode (corresponding to the TaskTracker generating the data). To avoid this, *TeraGen* uses one mapper per TaskTracker, with each mapper generating a 1024 MB (16 blocks) file.

Figure 5 shows the distribution of files across racks and tiers. To obtain this information, we parse and analyze HDFS's block map that stores information about all the blocks associated with a DataNode. For the network-aware policy, we find that more replicas of a file are placed in Tier-3 than in Tier-1 and Tier-2. This is because there are a fewer number of Tier-1 and Tier-2 DataNodes in comparison to Tier-3. In this policy, the distribution of replicas across the tier is directly proportional to the number of DataNodes contained in the tier. The results for the tier-aware and hybrid policies reveal replication of a block across all tiers. Since all the racks have the same number of nodes, even the network oblivious policies – random and tier-aware – have equal number of replicas across racks. The difference between these and the network-aware policy is that, in the later, each block is replicated across multiple racks to achieve resilience against rack failures. For the case of network oblivious policies, we see that 12% of the blocks are replicated only within one rack and will be exposed to data loss in case of a rack failure.

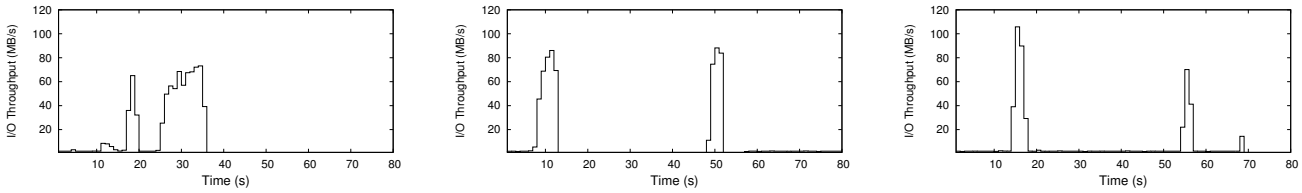
D. Impact of Retrieval Policy

In the next set of experiments, we study the role of the retrieval policies of hat*S*. We used *Grep* to read the data generated by *TeraGen* using six mappers per TaskTracker. Figure 6 shows the results. We observe that the random and the network-aware retrieval policies do not read a large number of files from the faster Tier-1 or Tier-2. This is mainly due to the fast tiers having a fewer number of blocks. Moreover, the probability that an access to a replica will be sent to a specific tier depends on the number of DataNodes in that tier, thus a tier with fewer blocks have fewer accesses. We find that the network-aware policy has 22% and 33% less remote requests than the random and tier-aware policies, respectively.

As expected, the tier-aware and hybrid policies access more requests from the fast tiers compared to the slow tiers. We observe that the hybrid policy results in $4\times$ more accesses to the Tier-1 than the network-aware policy, and only 13% more remote accesses than the network-aware policy. Further examination reveals that the hybrid policy results in 30% more accesses to Tier-1 and Tier-2, though at the cost of 15% increase in non-node-local (rack-local and remote) accesses. This trade-off between tier and network awareness offers an effective control knob that can be modified based on the infrastructure provisioning of a cluster to maximize performance.

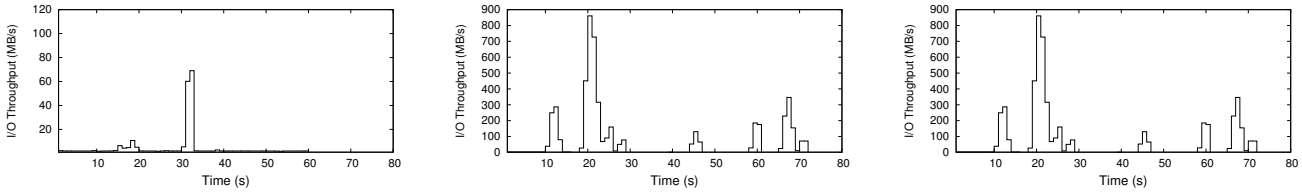
E. Impact of Network Speed on hatS Performance

In the next set of experiments, we compare the average I/O rate of the studied data management policies under our two testbed interconnects: 1 Gbps Ethernet and InfiniBand. In Figure 7, we see that the average I/O rate under InfiniBand



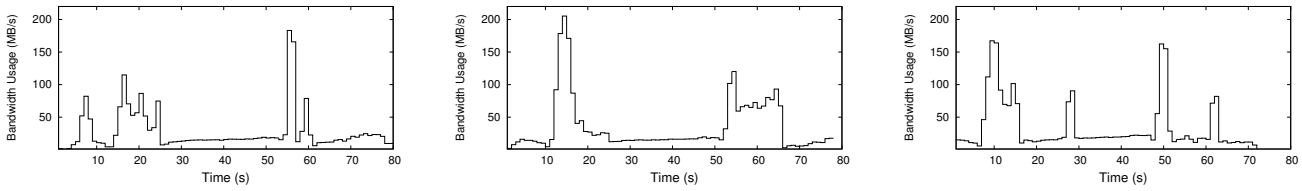
(a) DataNode with HDD. (b) DataNode with SATA SSD. (c) DataNode with PCIe SSD.

Fig. 9. Disk usage under the network-aware policy.



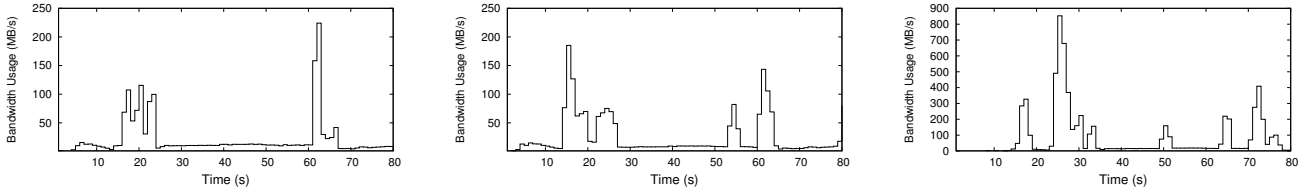
(a) DataNode with HDD. (b) DataNode with SATA SSD. (c) DataNode with PCIe SSD.

Fig. 10. Disk usage under the hybrid policy.



(a) DataNode with HDD. (b) DataNode with SATA SSD. (c) DataNode with PCIe SSD.

Fig. 11. Network usage under the network-aware policy.



(a) DataNode with HDD. (b) DataNode with SATA SSD. (c) DataNode with PCIe SSD.

Fig. 12. Network usage under the hybrid policy.

is better than that achieved under the 1 Gbps Ethernet. While expected, this result serves as a sanity check that our enhancements do not have unintended side-effects. Moreover, in the 1 Gbps Ethernet setup, we find that the network-aware policy performs better than other policies. We observe no additional advantages of the tier-aware policies with 1 Gbps Ethernet. In case of the random policy, the performance is better than the tier-aware policy. This is because of the network contention for the fast tier DataNodes as more requests are routed to them. From this test, we observe that better network provisioning is necessary to avail the benefits of hat*S*. However, this is not a limitation, as better interconnects are typical in enterprise data center deployments.

*F. Network and Disk Utilization in hat*S *DataNodes*

Next, we compare the network and disk usage of hat*S* for the read operation under two policies: network-aware and hybrid. For this purpose, we repeated the test described in Section V-D and used SAR [21] to collect detailed disk and network usage statistics for the DataNodes. Figures 9, 10, 11, and 12 show the behavior of one DataNode in each tier; similar patterns were observed for other DataNodes in the respective tiers.

As shown in Figure 9, under the network-aware policy, the HDD utilization is 40% higher than the combined utilization of the SATA SSD and the PCIe SSD. This highlights the disadvantage that the network-aware policy does not effectively utilize the expensive high performance storage devices. In contrast, Figure 10 shows the disk usage statistics under the hybrid policy. In this case, the number of requests to the DataNodes contained in Tier-3 are minimized. The SATA and PCIe SSDs together service 36% more read accesses than HDDs, and effectively utilize their high I/O bandwidth. Under hybrid policy, the utilization of PCIe SSDs has increased by 91%, and its maximum I/O throughput is 10$\times$ of that achieved under the network-aware policy. This shows the advantages of hat*S* over standard HDFS in better managing

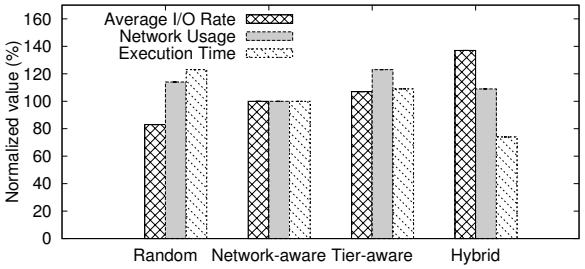


Fig. 13. Comparison of the average I/O rate, network usage, and execution time under the studied policies normalized to case of the network-aware policy.

the heterogeneous storage devices.

Similarly, Figure 11 and Figure 12 show the network throughput of DataNodes belonging to different tiers under the network-aware and hybrid policies. The network utilization is observed to be almost the same for DataNodes with the HDD and SATA SSD, whereas for DataNodes with the PCIe SSD the maximum bandwidth throughput is very high. This is because, under hybrid policy, three PCIe SSDs serve 28% more requests than under the network-aware policy, resulting in an increase in remote accesses.

G. Impact of Storage Characteristics on Hadoop Performance

In out next experiment, we study the impact of different storage devices on Hadoop. For this test, we provision HDFS to service all the requests from only one type of device for its storage. Our testbed contains only three PCIe SSDs but 27 HDDs, so as to ensure fairness and avoid performance bottleneck due to network contention for the SSD DataNodes, we reduce the number of worker nodes for this experiment to five nodes per rack. Each rack contains one PCIe SSD, three SATA SSDs and five HDDs. Figure 8 compares the execution time of *TeraGen* and *Grep* for three cases: HDFS with three PCIe SSDs, with six SATA SSDs, and 15 HDDs. *TeraGen*, which is a write-intensive application, performs better with HDDs and SATA SSDs than with PCIe SSDs. This is similar to our observation in Section V-B for *TestDFSIO-write*.

After each benchmark, we clear the contents of the DataNodes' buffer caches to prevent cross-benchmark pollution. For *Grep*, which involves a significant amount of read operations, we find that even though there is a small number of the PCIe SSDs, they perform significantly better than the SATA SSDs and HDDs. The three PCIe SSDs perform 20% faster than the 15 disks. We repeated the experiment with 21 worker nodes and found that the read performance of the three PCIe SSDs was similar to that of 21 HDDs.

H. Simulation-Based Experiments

For our next set of experiments, we developed an accurate simulator for hat*S* to observe the behavior of the considered data management policies on a large cluster setup. Our fine-grained simulator takes into account details such as the effect of intermediate shuffle data, network and storage infrastructure, and application I/O patterns. We simulated a 500-node cluster, with each node equipped with six 1 TB disks and one 256 GB PCIe SSD. These nodes are interconnected using two 10 Gbps InfiniBand links. We use the publicly available synthetic Facebook production traces [6] for driving the simulation. We replay the traces using HiBench [17] applications to process 70 TB of input data, generate 17 GB of intermediate shuffle data and 13 GB of output data spanning over three weeks. To make room for newly generated data, we use Least recently used (LRU) policy to evict the data.

Figure 13 compares the average I/O rate (the higher the better), network usage and trace execution time for the studied policies normalized to the case of the network-aware policy. We observe that the hybrid policy yields a 37% higher I/O rate as compared to the network-aware policy, and at the cost of 9% increase in the network usage (the lower the better). The tier-aware policy results in the highest network usage, i.e., 23% more than the network-aware policy. Data placement and retrieval behavior observed in the simulations is similar to that observed for the real testbed experiments. We see that overall SSD usage was improved by 68% with over 52% of the data accessed from the PCIe SSD. Finally, we also study the execution time (the lower the better) of the benchmarks under different policies. Our hybrid policy offers the best execution time, which is 26% better than the extant network-aware policy.

In summary, our evaluation of hat*S* reveals that it offers a viable solution to enhancing HDFS to incorporate heterogeneous storage, and does so efficiently. Our hybrid data management policy captures both tier awareness and network awareness to offer higher I/O rates and reduced execution time. These features are key to sustaining Hadoop for emerging architectures and applications.

VI. Related Work

Several recent projects [18], [40], [1], [14], [27] focus on tiered storage for general purpose enterprise computing, mainly due to its ease of management and business value. These systems typically employ SSD based tiering and caching, along with data management across tiers, to get higher I/O rates than just from HDDs. In hat*S*, we aim to extend such storage solutions to beyond individual nodes and servers, and into Hadoop's distributed setting. The recent HDFS-2832 [16] also calls for enabling support for heterogeneous storage in HDFS. hat*S* offers such support as well as provide different storage and data retrieval schemes to exploit the heterogeneity.

Spark [39] aims to avoid expensive HDD accesses by providing primitives for in-memory MapReduce computing. Our own Localized Storage Node (LSN) [20] proposes to divide a Hadoop cluster rack into several sub-racks, and consolidate disks of a sub-racks compute nodes into a separate shared LSN. MixApart [24] reduces the cost and inefficiencies of shared storage systems by offering a single consolidated storage back-end for managing enterprise data and servicing all types of workloads. AptStore [19] is a federated file system for Hadoop with a unified view of data from a multitude of sources, but stores all replicas of a file on one type of device.

These works share with hat*S* the focus on using tiered storage, but differ in that they do not support storage heterogeneity.

In the distributed setting, Zebra [15], GPFS [31], and Panasas [36], offer techniques to improve read and write throughput. Similarly, Flat Datacenter Storage [25] that employs an advanced network topology, and Camdoop [8] that uses a directly-connected network topology, argue the need for better provisioning and utilization of the disk bandwidth in the modern big data applications. These works are complementary to our design. Significant research has been done on network provisioning for Hadoop but incorporating fast storage technologies along with the traditional disks has not been previously explored.

VII. Conclusions

In this paper, we design and implement a novel heterogeneity-aware and tier-based enhancement for HDFS, hat*S*. Our solution also supports tier-aware data storage and retrieval policies to exploit the individual advantages of the various types of storage elements. We design easy-to-use APIs to allow movement of stored data across tiers, modify the number of replicas of a file in a tier, and monitor the capacity of the available tiers. Thus, hat*S* offers a flexible storage solution that can yield higher I/O performance by matching the application needs to appropriate storage tiers. We evaluate hat*S* using a range of representative applications and configuration parameters. Our analysis shows that, for the studied applications and setup, grouping storage devices in tiers according to their I/O bandwidth increases the utilization of the high performance storage devices by 91%, enabling them to service 64% more requests. This results in 32.6% improvement in throughput and 26% improvement in job completion time.

Acknowledgment

This work was sponsored in part by the NSF under Grants CNS-1016793 and CCF-0746832.

References

[1] Data storage on a multi-tiered disk system, Aug. 2 2005. US Patent 6,925,529.

[2] Apache Software Foundation. Hadoop, 2011. http://hadoop.apache.org/core/.

[3] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache hadoop goes realtime at Facebook. In *Proc. ACM SIGMOD*, 2011.

[4] L.-P. Chang. Hybrid solid-state disks: combining heterogeneous nand flash in large ssds. In *Proc. IEEE ASPDAC*, 2008.

[5] F. Chen, D. A. Koufaty, and X. Zhang. Hystor: making the best use of solid state drives in high performance storage systems. In *Proc. ACM SC*, 2011.

[6] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In *Proc. IEEE MASCOTS*, 2011.

[7] S. S. Chu and C. V. Ho. Self-recovering erase scheme to enhance flash memory endurance, Mar. 21 1995. US Patent 5,400,286.

[8] P. Costa, A. Donnelly, A. Rowstron, and G. OShea. Camdoop: Exploiting in-network aggregation for big data applications. In *Proc. USENIX NSDI*, 2012.

[9] J. Dean. Designs, lessons and advice from building large distributed systems. *Keynote from LADIS*, 2009.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[11] N. Dhondy and D. Petersen. Geographically dispersed parallel sysplex: The ultimate e-business availability solution. Technical report, IBM Corp, 2002.

[12] Dhruba Borthakur. Facebook has the world's largest Hadoop cluster!, 2010. http://hadoopblog.blogspot.com/2010/05/facebook-has-worlds-largest-hadoop.html.

[13] F. Dong. Extending starfish to support the growing hadoop ecosystem. Master's thesis, Duke University, 2012.

[14] S. Feldman and R. L. Villars. The information lifecycle management imperative. *IDC White Paper, July*, 2006.

[15] J. H. Hartman and J. K. Ousterhout. The zebra striped network file system. *ACM TOCS*, 13(3):274–310, 1995.

[16] HDFS-2832. Enable support for heterogeneous storages in HDFS, 2012. https://issues.apache.org/jira/browse/HDFS-2832.

[17] S. Huang, J. Huang, Y. Liu, L. Yi, and J. Dai. Hibench: A representative and comprehensive hadoop benchmark suite. In *Proc. ICDE Workshops*, 2010.

[18] H. Kario. Lvmts, 2012. https://github.com/tomato42/lvmts.

[19] K. Krish, A. Khasymski, A. R. Butt, S. Tiwari, and M. Bhandarkar. Aptstore:dynamic storage management for hadoop. In *Proc. IEEE CloudCom*, 2013.

[20] K. Krish, A. Khasymski, G. Wang, A. R. Butt, and G. Makkar. On the use of shared storage in shared-nothing environments. In *Proc. IEEE Big Data*, 2013.

[21] Linux man page. sar(1), 2013. http://linux.die.net/man/1/sar.

[22] R. Mantri, R. Ingle, and P. Patil. Scdp: Scalable, cost-effective, distributed and parallel computing model for academics. In *Proc. ICECT*, 2011.

[23] R. Micheloni, L. Crippa, and M. Picca. Hybrid storage. In *Inside Solid State Drives (SSDs)*, pages 61–77. Springer, 2013.

[24] M. Mihailescu, G. Soundararajan, and C. Amza. Mixapart: decoupled analytics for shared storage systems. In *Proc. USENIX FAST*, 2013.

[25] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. In *Proc. USENIX OSDI*, 2012.

[26] H. Payer, M. A. Sanvido, Z. Z. Bandic, and C. M. Kirsch. Combo drive: Optimizing cost and performance in a heterogeneous storage device. In *Proc. WISH*, 2009.

[27] M. Peterson. Ilm and tiered storage. Technical report, Storage Networking Industry Association, 2006.

[28] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Proc. USENIX FAST*, 2007.

[29] Sameer Tiwari. Hadoop and Disparate Data Stores, 2012. http://blog.gopivotal.com/products/hadoop-and-disparate-data-stores.

[30] Sameer Tiwari. Managing Hot and Cold Data Using a Unified Storage System, 2012. http://blog.gopivotal.com/products/managing-hot-and-cold-data-using-a-unified-storage-system.

[31] F. B. Schmuck and R. L. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proc. USENIX FAST*, 2002.

[32] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proc. IEEE MSST*, 2010.

[33] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending ssd lifetimes with disk-based write caches. In *Proc. USENIX FAST*, 2010.

[34] 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 *Proc. SOCC*, 2013.

[35] A.-I. Wang, P. L. Reiher, G. J. Popek, and G. H. Kuenning. Conquest: Better performance through a disk/persistent-ram hybrid file system. In *Proc. USENIX ATC*, 2002.

[36] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the panasas parallel file system. In *Proc. USENIX FAST*, 2008.

[37] T. White. *Hadoop: the definitive guide*. O'Reilly Media, 2012.

[38] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin. Improving mapreduce performance through data placement in heterogeneous hadoop clusters. In *Proc. IEEE IPDPSW*, 2010.

[39] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proc. USENIX HotCloud*, 2010.

[40] X. Zhao, Z. Li, X. Zhang, and L. Zeng. Block-level data migration in tiered storage system. In *Proc. IEEE ICCNT*, 2010.