AptStore: Dynamic Storage Management for Hadoop

Krish K. R.†, Aleksandr Khasymski†, Ali R. Butt†, Sameer Tiwari‡, Milind Bhandarkar‡
†Virginia Tech, ‡Greenplum
{kris,khasymskia,butta}@cs.vt.edu, {Sameer.Tiwar, Milind.Bhandarkar}@emc.com

***Abstract*—**
Typical Hadoop setups employ Direct Attached Storage (DAS) with compute nodes and uniform replication of data to sustain high I/O throughput and fault tolerance. However, not all data is accessed at the same time or rate. Thus, if a large replication factor is used to support higher throughput for popular data, it wastes storage by unnecessarily replicating unpopular data as well. Conversely, if less replication is used to conserve storage for the unpopular data, it means fewer replicas for even popular data and thus lower I/O throughput. We present AptStore, a dynamic data management system for Hadoop, which aims to improve overall I/O throughput while reducing storage cost. We design a tiered storage that uses the standard DAS for popular data to sustain high I/O throughput, and network-attached enterprise filers for cost-effective, fault-tolerant, but lower-throughput storage for unpopular data. We design a file Popularity Prediction Algorithm (PPA) that analyzes file system audit logs and predicts the appropriate storage policy of each file, as well as use the information for transparent data movement between tiers. Our evaluation of AptStore on a real cluster shows 21.3% improvement in application execution time over standard Hadoop, while trace driven simulations show 23.7% increase in read throughput and 43.4% reduction in the storage capacity requirement of the system.

I. Introduction

Hadoop Distributed File System (HDFS) [27] provides a robust storage for managing massive amounts of data in a scalable manner by aggregating the direct attached storage (DAS) of Hadoop cluster nodes [7]. The off-the-shelf machines that make up typical Hadoop clusters and the scale of the system imply that failures are the norm. To prevent data loss, HDFS relies on replication [27]. Replication also increases the read throughput, not only because it reduces access contentions that can arise when accessing popular data, but also by increasing the probability of finding the data on a local DAS.

While DAS with replication offers significant throughput benefits in Hadoop, the default three replicas also incur a 200% storage overhead. Not only does this overhead add to the direct cost of the storage, it has indirect maintenance costs of energy consumption and administration, which can be significant [22]. Another limitation of the DAS-based Hadoop architecture is that storage capacity is tightly coupled with compute capacity; to add more storage, more compute nodes need to be added. Thus increasing storage capacity in standard DAS-based Hadoop also incurs the cost for compute components, which may be unnecessary for typically I/O-bound Hadoop applications. Adding a whole node for just using the extra storage exacerbates energy efficiency as well, as typically, storage accounts for only a fraction of a Hadoop node's energy consumption [28].

To this end, Network Attached Storage (NAS) can offer an alternate storage solution for Hadoop, especially enterprise NAS is attractive due to its lower failure rates. To add to this, the per GB storage cost in enterprise storage solutions [25] is only a fraction of that in a commodity Hadoop node DAS. However, the challenge is that naively adding NAS to Hadoop clusters may entail a large number of data accesses over the network, resulting in reduced I/O throughput. A promising trend observed in recent analysis is that there is significant heterogeneity in I/O access patterns. GreenHDFS [16] observed a news server like access pattern in HDFS audit logs from Yahoo, where recent data is accessed more than the old data and more than 60% of used capacity remains untouched for at least one month (period of the analysis). Scarlett [4] analyzed job history logs from Bing production clusters and observed that 12% of the most popular files are accessed over ten times more than the bottom third of the data.

In this paper, we design a tiered storage system, AptStore, with two tiers designed to better match the heterogeneous Hadoop I/O access patterns. The tiers include: Primary storage — DAS in Hadoop node for files that require high throughput; and Secondary Storage — NAS for unpopular files and files with lower Service Level Objectives (SLO). AptStore analyzes the I/O access patterns and suggests data placement policies across the tiers to increase the performance and efficiency of the storage system. Our system optimizes for read throughput as typically MapReduce workloads exhibit write-once read-many characteristics [27]. To achieve this, we predict the popularity of each file, and then retain the popular files in primary storage and move unpopular files to secondary storage. We also adjust the replication factor of files in primary storage based on their popularity. The replication factor for files in the secondary storage is set to 1, and other means such as RAID are employed to achieve fault tolerance. We have realized AptStore as an extension to the Unified Storage System (USS) [1], [25], a federated file system for Hadoop, which allows transparent movement and management of data across different file systems.

Specifically, this paper makes the following contributions:

- We present a detailed quantitative study of the factors that affect the read throughput in HDFS, such as block and file size, replication factor, locality, and number and frequency of concurrent accesses to files.

- We design and implement the tiered storage solution of AptStore, which improves both storage efficiency and read throughput.
- We design an access pattern based popularity prediction algorithm (PPA) that predicts popularity of files based on file size, access frequency, and load in the cluster.
- We validate and evaluate our data placement strategy using both trace-driven simulations and experimentation on a real cluster.

Evaluation of AptStore shows a 43.4% average reduction in the disk space requirement and 23.7% increase in the read throughput over standard Hadoop framework in our simulations. Moreover, our implementation of AptStore achieves up to 21.3% speed up in studied applications execution time compared to standard Hadoop.

II. Factors Affecting Hadoop Storage Performance

In the following, we discuss the key factors that impact the performance and efficiency of the storage system in Hadoop.

A. Understanding Read Throughput

There are two key factors that affect read throughput in HDFS: data locality and number of concurrent accesses. Local accesses result when a job and its associated data reside on the same node, thus reducing the number of remote I/O requests and yielding higher throughput. On the other hand, many concurrent accesses to the same file increase contention, thus decreasing read throughput.

1) Locality: HDFS divides the data into equal sized blocks and distributes data to multiple nodes, which distributes the read request throughout the cluster, thereby achieving better aggregate throughput. Block size is an important tunable parameter in the system. Bigger block sizes decrease the overall number of blocks per file and hence the number of nodes that hold data. This decreases probability of the job scheduler assigning tasks that are local to the data. However, decreasing the block size too much is also undesirable as it can result in memory contention in the *NameNode*. With constant block size, file size has a direct effect on the distribution of the data; larger files are distributed throughout the cluster, while the smaller files are restricted to a small set of nodes.

Replication also affects locality and in turn the read throughput in a Hadoop cluster [27]. Higher replication factor increases the distribution of the data in the cluster, thereby increasing the probability of *JobTracker* finding a local or rack local slot for a task. This results in reduced network and disk contention, particularly when multiple jobs access the same data concurrently. Thus, a small file with more replicas can have the same distribution as that of a larger file.

2) Concurrent Access: Concurrent jobs accessing a single block, or blocks from a single machine, not only affect the disk bandwidth available per access, but also the network bandwidth. In Hadoop clusters, concurrent access to popular data are common [4]. In such cases, when the number of tasks accessing the data exceeds the number of replicas, read throughput of the tasks is affected because of slot contention and hardware resource contention[5].

Large number of concurrent tasks reading data from a node decrease the probability of scheduling a task local to the data, and as a result read throughput includes network overhead. Since concurrent requests share the disk bandwidth and network bandwidth, the number of concurrent accesses is inversely proportional to the read throughput. The scenario is typical in production clusters, especially on machines storing popular data. Scarlett's [4] analysis of Bing production cluster indicates that more than 50% of read requests were directed to less than 17% of the cluster. The decrease in read throughput because of increased concurrent accesses can be reduced by increasing the replication of the file and distributing the requests across the cluster.

B. Fault Tolerance

Since Hadoop clusters are built using commodity machines, the hardware failure rate is non-negligible. The typical Mean Time Between Failures (MTBF) is 3 years [6], so for a thousand node Hadoop cluster, the probability of failure of a single machine in the cluster is close to one. Data loss prevention using RAID is not a feasible solution because equipping each Hadoop node with a RAID controller is expensive and software RAID on unreliable machines incurs high performance overhead. Since availability is proportional to MTBF, reducing the replication factor to 1 and using parity to prevent data loss might result in reduced availability. Moreover the low reliability of the hardware implies periodic loss of data resulting in reconstruction from the parity. Such generation and reconstruction will adversely affect the performance of the in-progress Hadoop jobs.

In contrast, a more feasible RAID based solution can be used by employing consolidated NAS if high I/O throughput is not a concern. Enterprise storage solutions typically utilize RAID and have lower MTBF [11]. These devices ensure the same reliability of data with significantly less storage overhead. Moreover these devices are self managed and reconstruction of parity would not affect the in-progress Hadoop jobs, unless the jobs are trying to access the files under recreation. Thus, incorporating NAS into Hadoop architecture is promising and can support low-cost fault-tolerant storage.

C. Storage Cost

The use of replication increases the capacity needed to be provisioned and thereby exacerbates the cost associated with the storage. While DAS offers better performance at higher cost, enterprise filers offer degraded performance at lower cost. Thus, it would be beneficial to utilize the different kinds of storage in realizing an efficient Hadoop storage architecture, provided the performance requirements for the data items can be determined or predicted.

Typically, a Hadoop node with a maximum of 24 TB of data storage uses up to 200 W [2] at idle state. The energy cost of adding a Hadoop node for storage scalability would result in 8.33 W/TB. Along with the 200% storage overhead the

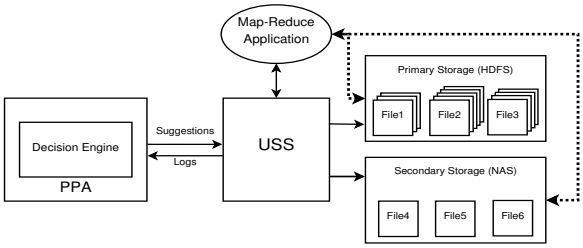


Fig. 1. AptStore architecture overview.

energy cost of storing data in DAS is 25 W/TB. This is very high when compared to the 0.78 W/TB in enterprise storage solutions [19]. Thus, from the energy consumption perspective, use of NAS in Hadoop is very favorable.

III. System Design

In this section, we describe the design of AptStore, including its decision engine, the Popularity Prediction Algorithm (PPA), as well as how AptStore is realized within an available enterprise NAS implementation.

A. AptStore

AptStore is designed as an extension to the Unified Storage System (USS) [1], [25] [1], a federated file system for Hadoop, which allows tiered storage across different file systems. As access rate and number of accesses varies for each file, AptStore improves the overall read throughput and storage efficiency of the system by designing access-pattern-based data placement and replication. Figure 1 illustrates the components of AptStore and their interactions. The PPA periodically analyzes the usage patterns of the file system and the Decision Engine (DE) suggests appropriate data placement strategy.

In designing AptStore, we make the following four design choices. First, we consider replication at file granularity because Hadoop jobs access files as a whole [4]. Making replication decisions at the block level is unnecessary, as the read cost of a file will be dependent on the block with the lowest replication factor. Second, we assume that typical production clusters are heavily used and have very large working sets. Thus, any effects of file system level caching is negligible, and the majority of reads are serviced from disk. Third, all the files in the system are assumed to have the same block size, which is standard in Hadoop deployments. Finally, AptStore is designed for the extant Hadoop deployments where all nodes contribute storage and computation.

B. Unified Storage System

The Hadoop framework works best when used in conjunction with HDFS. The Hadoop command line (`FsShell`) and file system API are supported only by HDFS and its variants [23]. Jobs involving data from other sources are processed by first loading the needed data into HDFS, typically by using tools such as `cron`, `scp`, and `distcp`. Another solution to multi-source data access is to use viewFS [14] or add data sources directly using Uniform resource identifiers (URI). However, adding enterprise storage devices through these approaches lead to load imbalance and decreased read throughput, mainly because the devices would likely not be available centrally/equidistant across the cluster.

To address such issues, USS implements a federated file system that provides a unified view using a single namespace that encompass a multitude of data sources. USS supports transparent, zero-copy access of data from various data sources. It also maintains a mapping of all HDFS files to their actual locations in the respective file systems. We leverage and extend this feature in AptStore to transparently move data between primary and secondary storage as needed.

C. Popularity Prediction Algorithm

We design a Popularity Prediction Algorithm (PPA) using file access information to determine when and where to store the files. At every RT, the PPA analyzes the access pattern for each file and predicts a expected popularity value for it for the next RT. The popularity value $P_{i+1}(f)$ of a file f varies with each access $i+1$ to the file. $P_{i+1}(f)$ is defined as:

$$P_{i+1}(f) = P_i(f) + \frac{c}{a(f) * l * b(f) * P_i(f)}, \qquad (1)$$

where c is the popularity constant, $a(f)$ is a function of the access interval of file f, l is the load in the cluster and $b(f)$ is a function of number of blocks in the file f. Observe that we designed our popularity measure to recursively depend on the file popularity during the previous time interval. This causes the number of replicas to remain stable even in the presence of a bursty access patterns between successive intervals, yet adapt to changes that are longer lasting. Additionally, this allows the system to adapt effectively to access patterns of periodic jobs or ones that are scheduled at intervals wider than RT.

Equation 1 also ensures that the popularity of file f increases not only with the number of accesses but also if it is accessed concurrently by many clients. The access frequency is inversely proportional to the time between the previous access, i, and the current access, $i+1$. During any RT, files with the same number of accesses may have different access frequencies. For example, a file can have one access every five minutes for a total of 12 accesses in an hour, whereas another file may have 12 concurrent (non-repeating) accesses. Reads with higher access frequency require more replicas of the accessed files than those exhibiting lower accesses frequencies, even if the total number of reads within a RT are the same. This is because frequent reads cause contention both at the disk and network, resulting in degraded read throughput.

The required replication factor also depends on the cluster load, l, computed using the overall popularity of all files in the system. Many concurrent requests for multiple files can compound and result in an increase in contention for both the disk and the network bandwidth. Although increasing the cluster infrastructure to handle a higher load is one possible solution, it is not always feasible. Our solution is to aggressively replicate popular data, because it would better distribute the requests across the cluster and increase the probability of accessing the data locally. Conversely, we reduce the number of replicas for unpopular data.

[1]Note that techniques developed in AptStore are not USS-specific and can be easily implemented and integrated with other NAS solutions.

As we assume a constant block size, larger files have more blocks. Consequently, larger files are better distributed throughout the cluster, so they require fewer number of replicas than smaller files. To capture this aspect, we update the popularity of the file after each access by an increment that is inversely proportional to the file size.

During the creation of a file, the popularity of the file $P_1(f)$ is initialized to average file popularity observed in the system, $AVG(P)$. Initialization based on observations such as the type of jobs accessing the file or popularity of other data created by the same user are also promising, but we leave that for future work. Similarly, whenever a file is deleted, it will result in popularity of other files being modified when the values are updated at the end of RT. When a popular file is deleted, the popularity of other files in the system increases. Conversely, when an unpopular file is deleted, the popularity of other files decreases. We do fix the minimum, P_{Min}, and maximum, P_{Max}, threshold on the popularity of a file to make sure that there are bounds on the number of file replicas in the system. The minimum threshold ensures data reliability and compliance with system SLAs, while maximum threshold captures space constraints in primary storage.

After the accesses of all files in the reference time RT are processed the popularity value $P_i(f)$ of a file f for the most recent access i is modified as follows:

$$P_i(f) = P_i(f) - \frac{MIP}{s}, \tag{2}$$

where MIP is the mean increase in the popularity of the file f during reference time RT, $AVG(P)$ is the average popularity of all the files in the cluster, s is the scalability constant. Equation 2 ensures that the popularity of the file $P(f)$ does not grow arbitrarily. The mean increase in popularity is a fraction of increase in popularity, IP during RT over F, the set of all files in the system. The scalability constant s, is used to contract or expand the amount of data stored in primary storage. For a value of s greater than one, more data is pushed to primary, while a positive value of s, less than one, creates more space in the primary storage.

The choice of RT is critical. A very large RT can miss opportunities to change the file replication factor to adapt to a change in the access pattern as the workload varies. However, setting RT too small can result in excessive thrashing as PPA state is rapidly updated. The appropriate value of RT depends on the usage pattern of the cluster and the cluster infrastructure. Previous work [4] suggest an RT between 12 and 24 hours is sufficient to capture varying patterns in the workload, while minimizing overheads associated with managing extra replicas.

Finally, AptStore adopts a proactive prediction scheme. The predicted popularity, $P_{predicted}(f)$, for the the next RT is computed using a linear extrapolation. We assume that the rate of change of popularity for the next RT will be same as the rate of change of the current RT. By choosing an appropriate size of RT, the accuracy of the predicted popularity can be made high.

Input : USS file System Audit Logs
Output: Predicted popularity $P_{predicted}$.
F *is the set of files in the file system*;
foreach *access* $i+1$ *to the file* $f \in F$ *in* RT **do**
  if i==*0* **then**
    $P_{i+1}(f) \leftarrow AVG(P)$;
  end
  else
    $P_{i+1}(f) \leftarrow P_i(f) + \frac{c}{a(f)*l*b(f)*P_i(f)}$;
  end
  if $P_i(f) < P_{Min}P$ **then** $P_i(f) \leftarrow P_{Min}$;
  else if $P_i(f) > P_{Max}$ **then** $P_i(f) \leftarrow P_{Max}$;
  $IP = IP + P_{i+1}(f) - P_i(f)$;
end
foreach *deletion of the file* f *in* F **do**
  $IP = IP + AVG(P) - P(f)$;
end
$MIP \leftarrow \frac{IP}{size(F)}$;
foreach *file* f *in* F **do**
  $P_i(f) \leftarrow P_i(f) - \frac{MIP}{s}$;
  where i is the most recent access to the file f.
  $P_{predicted}(f) \leftarrow P_i(f) + (P(f) - P_i(f))$;
  $P(f) \leftarrow P_i(f)$;
end

Algorithm 1: Popularity Prediction Algorithm.

D. Decision Engine

At every reference time RT, the decision system suggests the replication and data placement strategy for a file f based on $P_{predicted}(f)$, the predicted popularity of the file for the next RT . Files with higher popularity are replicated based on the function $P_{predicted}(f)$ and are placed in the primary storage. Files with lower popularity are moved to secondary storage and the replication factor is reduced to 1. Files with average popularity are maintained in the primary storage with default replication levels.

The system also considers cron jobs or jobs that are scheduled for later execution and predicts the appropriate storage strategy, thereby improving the read throughput. Finally, the system considers the SLA requirement irrespective of the popularity. For example, an unpopular files, although accessed rarely, may have significant SLA restriction, so it may be always replicated and stored in the primary storage.

E. Replication and Inter-tier Data Movement

Hadoop performance is sensitive to network bandwidth. Replication or data movement across tiers during such network intensive phase may adversely affect the performance of the Hadoop jobs in progress. To balance the bandwidth consumption, HDFS employs multi-location replication [4], where the data to be replicated are read from multiple sources thereby spreading the replication traffic across multiple nodes. File movement between primary and secondary storage as well as change in replication factor is realized by a low priority background process.

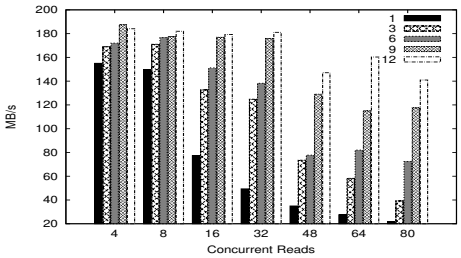


Fig. 2. Read Bandwidth with increasing replication factor and with increasing number of concurrent reads.

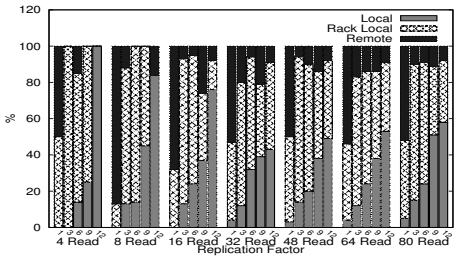


Fig. 3. Number of Local vs. Rack Local vs. Remote with increasing replication factor and with increasing number of concurrent reads.

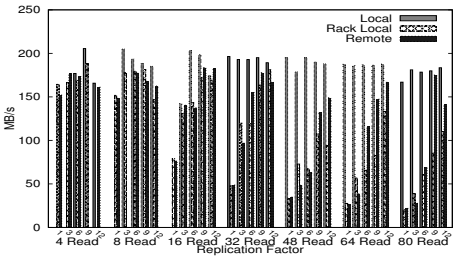


Fig. 4. Bandwidth of Local vs. Rack Local vs. Remote with increasing replication factor and with increasing number of concurrent reads.

System performance may further increase with a rack-level dedicated link from primary storage to secondary storage. Furthermore, significant work is done to improve the network utilization and storage utilization in Hadoop by compressing the data [10], which can be leveraged. For replica deletion, lazy deletion [4] of data, i.e., waiting for it to be overwritten by another block, may significantly reduce the cost of deletion.

F. AptStore Computation Overhead

AptStore requires calculation of each file's popularity at the end of every RT. This computational overhead is negligible, because popularity is computed by linearly processing the file system audit logs, which the NameNode already generates. Hence, from the point of view of the TaskTrackers, our system produces negligible overhead for typical cluster sizes. For small clusters, PPA algorithm can run on the same node as the NameNode. However, for a very large cluster where popularity computation may incur some overhead, the algorithm can be offloaded to a separate machine.

The design of AptStore aims to monitor file I/O and utilize the PPA to determine the popularity of individual files. Our system extends USS to use the popularity information to move the files between primary and secondary storage tiers, thus providing high replication and high throughput for popular data, and low-overhead high-volume cheaper storage for unpopular data. Moreover, the use of PPA ensures that the AptStore is able to adapt to the changing characteristics of the Hadoop workloads.

IV. Evaluation

In this section, we present a detailed evaluation of factors that affect read performance in Hadoop and evaluate the performance of AptStore.

A. Experimental Setup

We use a 28 node cluster for our experiments. The master nodes have 3.33 GHz 2×Intel Xeon X5680 6-core CPUs with hyper-threading, 64 GB of RAM, and up to 3 SATA disks. The worker nodes have 3.33 GHz 2×Intel Xeon X5680 6-core CPUs with hyper-threading , 48 GB of RAM, and 12∗600, 15 K RPM disks. The master and workers are equipped with four and two network ports, respectively, and are interconnected using 10 Gbps link. There are two master nodes, one running a dedicated *NameNode* and the other the *JobTracker* and the *SecondaryNameNode*. Moreover, there are 26 worker nodes, each with an instance of *TaskTracker* and *DataNode*. The default HDFS block size is 512 MB and the version of Hadoop we employ is GPHD 1.2.

B. Impact of Design Parameters

In the first set of experiments, we analyze how various factors impact read throughput of Hadoop, and quantify the impact under different test conditions. To compute the read cost and eliminate any computation cost, we run a map-only job that reads a block of data. We execute this job with varying number of concurrent reads and on data sets of same size with varying replication factor. To minimize the effect of caching, we flush the file system caches on all disks contributing to HDFS between test runs.

1) Read Bandwidth Comparison: We observe the read bandwidth by varying the replication factor and the number of concurrent reads. Figure 2 shows the results. We find that 3 replicas provide sufficient for workloads with up to 8 concurrent reads and adding an more replicas produces marginal improvement in read throughput. As we increase the number of concurrent reads and thus the contention, more replicas are required to sustain the read bandwidth. For a workload with 80 concurrent accesses, any replication below 9 suffers significant loss in throughput. Conversely, in a workload with up to 32 concurrent accesses, increasing the number of replicas beyond 9 produces no performance benefit, thus wasting storage space used by the extra replicas. The results show that using a uniform replication factor is problematic in terms of both meeting throughput demands for popular files, and conserving space for unpopular files.

2) Impact of Access Locality: Next, we repeat the previous experiment but study the percentage of read requests that are served locally versus remotely. The results are shown in Figure 3. The percentage of local accesses increases with the replication factor. While the dataset with a replication factor of 12 achieves more than 40% of local accesses in all of the concurrent reads, using a replication factor of 3 achieves only a maximum of 14% local accesses.

Bandwidth of local, rack local, and remote access are compared in Figure 4. We find that the local access use similar bandwidth across varying concurrent number of reads. With increasing concurrent accesses, there is a difference between bandwidth for the rack local and remote rack accesses. For higher number of concurrent reads, files with lower replication factor shows low bandwidth. When remote requests are serviced, the network bandwidth is shared among the requests. With fewer number of replicas and high concurrent access, the contention for network resource of the nodes containing the replica is high, leading to low available bandwidth per access.

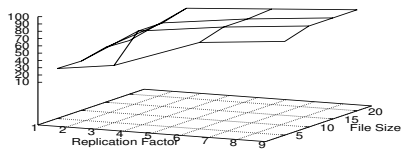


Fig. 6. Percentage of local tasks scheduled, with increasing file size and increasing replication factor.

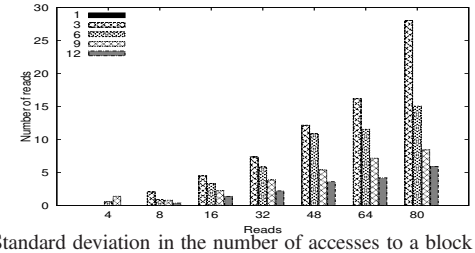


Fig. 7. Standard deviation in the number of accesses to a block of data while increasing the replication factor and the number of concurrent reads.

3) Impact of Access Variation: Not all replicas of a file are accessed equally. Whenever a client requests the *NameNode* for accessing a block, the *NameNode* returns the location of all the replicas of the block. The location list is ordered by its proximity from the requesting client. The client checks the availability of a local replica and if it fails to find one, it looks for a rack local, and then a remote rack occurrence of the block. If there are many more concurrent accesses than number of replicas, the scheduler is unable to balance the load to all replicas, because Hadoop performs scheduling on a best effort basis. Figure 7 shows the standard deviation in the number of accesses to a block of data with varying number of requests to the data and with varying replication factor. A workload with 3 replicas and 80 concurrent accesses produces very high variance, while increasing the replication produces a more uniform access pattern and thus lower variance. Such uniform access increases the overall throughput because the load in the system is more balanced and the contention for hardware resources is reduced.

4) Impact of Replication and File Size: Locality increases with the distribution of data in the cluster. Figure 5 shows that, as with increasing replication, increasing file size increases the data distribution among the cluster nodes, resulting in reduced contention and increased locality. Figure 5(a) shows that data with a replication factor of 1 is available only in one rack, but as the replication increases to 6 we find that data is more evenly distributed among racks. Figure 6 shows that with increasing file size and with increasing replication more and more local tasks are scheduled. Previous studies has shown similar behavior with MapReduce benchmarks such as TeraSort [16]. These results are promising as our PPA takes into account such behavior to provide accurate predictions.

C. Fault Tolerance in Hadoop

To study the effect of replication on overall fault tolerance in Hadoop, we first choose an appropriate failure model. We base our probability of data loss model on a previous study [26] and define $P_{data-loss}$ as:

$$P_{data-loss} = 1 - \sum_{f=0}^{n} P_{failure}(n, f) * P_{no-loss}(n, b, r, f), \quad (3)$$

TABLE I
PROBABILITY OF DATA LOSS PER DAY.

File System	Replication Factor	Number of nodes	$P_{data-loss}$ per day
HDFS	3	1000	$6.44 * 10^{-2}$
HDFS	3	100	$1.1 * 10^{-4}$
HDFS	2	1000	0.23
HDFS	2	100	$3.86 * 10^{-3}$
HDFS	1	1000	0.59
HDFS	1	100	$8.7 * 10^{-2}$
Filer	3	1000	$1 * 10^{-2}$
Filer	3	100	$1.42 * 10^{-5}$
Filer	2	1000	$7 * 10^{-2}$
Filer	2	100	$9.7 * 10^{-4}$
Filer	1	1000	0.36
Filer	1	100	$4.4 * 10^{-2}$

$$P_{failure}(n, f) = \binom{n}{f} * p^f * (1-p)^{(n-f)}, \quad (4)$$

$$P_{no-loss}(n, b, r, f) = (1 - \binom{f}{r} / \binom{n}{r})^b \quad (5)$$

where p is the probability of failure of a single machine, n is the number of machines in the cluster, r is the replication factor of a block, $P_{failure}(n, f)$ is the probability that there are exactly f failures in the cluster and $P_{no-loss}(n, b, r, f)$ is the probability that there is no data loss in the cluster [26]. Moreover, the probability of losing a node in time T, is $1 - R$, where R is the reliability of a node or the probability that the node will not fail over the time T. R is defined as:

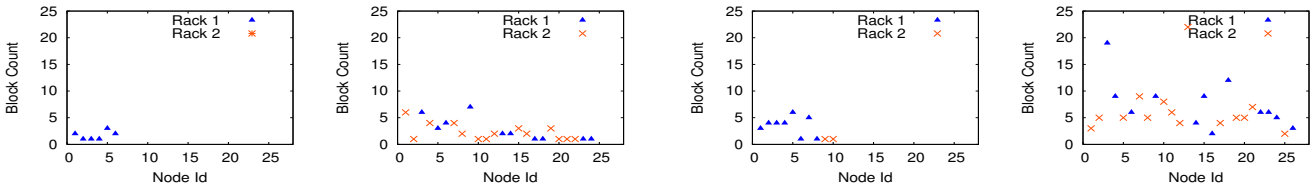
$$R = e^{-\frac{T}{MTBF}} \quad (6)$$

The MTBF of a Hadoop node and an enterprise storage server is three and six years respectively [6], [20]. Based on these values, Table I compares the probability of data loss per day in HDFS and NAS filers using Equations 3. We compare the HDFS cluster with 1000 nodes to a filer with 100 nodes assuming that they can offer the same storage capacity. This is valid assumption given recent trends in storage capacity. Enterprise systems can easily support more than 240 TB of storage [24], while a typical Hadoop node has 12 TB to 24 TB of storage.

Filers can offer probability of data loss with one replica of $4.4 * 10^{-2}$, compared to HDFS with 3 replicas, which has a $6.44 * 10^{-2}$ probability of data loss. Given the high number of disks in a single enterprise storage node, fault tolerance is handled by a RAID controller with a probability of data loss of $4.4 * 10^{-3}$, and hence in this case we decrease the replication factor to 1. Disks are arranged in, for example, a $(10, 2)$ RAID array, which protects from a simultaneous loss of two disks with ten-fold decrease in storage overhead when compared to two replicas. Thus, the use of filers as secondary storage is promising in AptStore and offer a most cost-efficient yet robust solution.

D. Performance Analysis of AptStore

In the next set of experiments, we evaluate AptStore both in a real system as well as using whole-system simulation.



(a) 5 GB file with replication factors 1 and 6.

(b) 10 GB file with replication factors 1 and 6.

Fig. 5. Data distribution in Hadoop with increasing file size and increasing replication factor.

Fig. 8. Comparison of read throughput between Hadoop, Scarlett, AptStore-perf and AptStore.

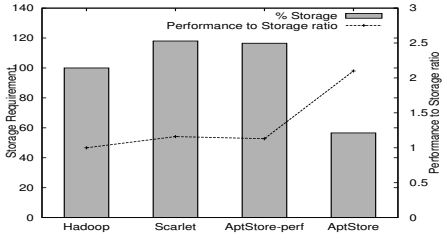


Fig. 9. Comparison of storage requirement between Hadoop, Scarlett, AptStore-perf and AptStore.

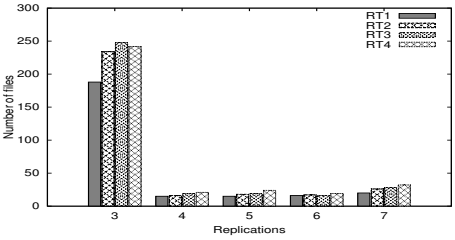


Fig. 10. Number of files with different number of replicas.

1) Experiments Using Trace Driven Simulation: For our simulation, we replay Facebook-like traces synthetically generated by sampling historical MapReduce cluster traces. The traces provided by Chen et. al. [9] are one day in duration and contain 24 historical trace samples each 1 hour long.

We use these traces to compare the performance of AptStore with the default Hadoop and an emulated version of Scarlett [4]. Scarlett is a budget-based Hadoop system that increases throughput by replicating files based on their access patterns. For every file, Scarlett computes the maximum number of concurrent accesses (cf) in a learning window of length TL. Once in every rearrangement period, TR, Scarlett computes desired replication factors for each file as $\max(cf + \delta, 3)$, where δ is the cushion factor against under estimation. In our simulation there were 24 rearrangement periods, i.e., rearrangement happens once every hour and our learning window is the same as that of the rearrangement period. We simulate various factors including the impact of replication, contentions while accessing popular data, and advantages of distribution of data.

The comparisons are made with two variants of our system. The first version, AptStore-perf stores file only in HDFS and the minimum replication factor of both popular and unpopular data is 3, while the maximum factor varies based on the popularity of the data. The second version, AptStore, uses two underlying file systems. The popular files reside in HDFS with varying replication and the unpopular files are pushed to a lower-throughput reliable storage with only one replica. AptStore-perf isolates and measures the performance gains of our techniques, while AptStore provides insight into both the performance and storage efficiency. Similar to Scarlett, our simulation of AptStore rearranges the data once every hour.

Figure 8 shows the comparison in the read throughput normalized to default Hadoop. Scarlett produce a 19.57% improvement in performance while AptStore-perf and AptStore produce 23.74% and 18.64% improvement over default Hadoop, respectively. It is important to note that in AptStore, where the popular files are fetched from primary storage and the unpopular files are fetched from secondary storage, only 2.6% of accesses are served by the secondary storage.

In the next experiment, we compare the storage requirement, shown in Figure 9, where the required storage is normalized to the case of default Hadoop. Scarlett requires 13% additional storage while AptStore-perf uses only 10% additional storage to achieve the same performance. AptStore achieves the same performance using only 57% of the storage required under standard Hadoop. The replication required for unpopular files in the secondary storage is considerably low when compared to the 200% storage overhead of the primary distributed file system. Figure 9 also compares the performance to storage ratio, normalized to default Hadoop and AptStore achieves a significant $2\times$ improvement over default Hadoop. Over the default Hadoop, Scarlett shows a 5% improvement in the ratio and AptStore-perf shows 12.5% improvement.

2) Experiments on a Real Testbed: We implemented AptStore on top of USS [1], [25]. Since all file system requests are handled by USS, the USS audit logs record all the access to the underlying file systems. The Hadoop master node communicates with AptStore, which provides it with a data management and replication strategy. Note that for very large clusters our system can run on a cluster of machines to compute the data placement strategy. AptStore accesses the logs and the PPA assigns a popularity to each file at the end of every reference time. The decision system gives hints to the USS for the appropriate replication policy for the file. Migration is performed through POSIX-like USS file system API, while replication of files in HDFS uses the `setrep` file system API in Hadoop. Our workload is generated from traces mentioned in Section IV-D1. We replace the jobs in the trace with `sort`, `grep` and `wordcount` [15]. We believe this approximation is reasonable as the advantage of AptStore is mainly because of read access in the map phase. For our implementation, we adjust the length of the trace and the size of the files to match the size of our test cluster. We did not have access to an enterprise filer, so we used HDFS for all the data, irrespective of their popularity. Our implementation shows that AptStore reduces the execution time of the trace by 21.9% over default Hadoop, with 11.9% increase in storage.

Figure 9 compares the number of files with different number of replicas. We observe that increasing the replication of 19% of files, results in a performance improvement of 21.9% over default Hadoop. The increase in the replication factor of certain files does not pertain to one single factor, it is based on the combination of factors described in Section III-C.

Our evaluations show the analysis of various factors affecting the read throughput and fault tolerance in HDFS and enterprise storage solution. We also show the impact of these factors on the design of AptStore, by comparing the performance of AptStore to Hadoop and Scarlett.

V. Related Work

Research on increasing the storage efficiency in GFS [13] and HDFS managed clusters [12] propose to asynchronously compress the replicated data down to RAID-class redundancy. However, these techniques lower MTBF, which results in lower availability and reliability. Much of the recent work focuses on energy efficiency in Hadoop storage [16], [17], [18], [3]. These works propose energy aware date placement, where unpopular data is placed in a subset of Hadoop cluster nodes, generating significant periods of idleness to operate in a high-energy-saving mode without affecting nodes containing the hot data. However, this approach increases the skewness in popularity as hot data is concentrated on a subset of nodes, resulting in degraded throughput compared to spreading the hot data throughout the entire cluster. To improve scalability MixApart [21] uses an integrated data caching and scheduling solution to allow MapReduce computations to analyze data stored on enterprise storage systems.

The work most similar to our own is Scarlett [4]. One shortcoming of this approach is that larger files are given a priority for increased replication over smaller files, and hence popular small files may still suffer read throughput degradation and popularity skewness. To achieve similar goals Yahoo proposed HotROD [24], an on-demand replication scheme, which allows access to the data from some other HDFS cluster by creating a proxy node. However, this approach might suffer degraded performance if inter-cluster network bandwidth is low. Finally, SCADS Director [8] offers a control framework that reconfigures the storage system on-the-fly in response to workload challenges using a performance model of the system, and in that is complementary to AptStore.

To the best of our knowledge, AptStore is the first work that addresses both storage efficiency and read performance.

VI. Conclusion

In this paper, we presented a dynamic data management scheme for Hadoop for achieving higher throughput and lower storage cost. We observe that managing all Hadoop data in a uniform manner results in increased storage overhead or reduced read throughput. For popular files, default replication is insufficient and leads to decreased throughput. For unpopular files, default replication results in storage inefficiency. We proposed AptStore, a system that exploits the heterogeneity in access patterns to achieve overall reduction in storage cost and increase in read throughput. We identify various factors that affect the throughput of the system and propose PPA to predict the popularity associated with each file, and use the information to adjust the replication and data placement strategy of the files. Using extensive simulations and a real deployment, we demonstrated that AptStore data management scheme increases the read throughput by 23.7%, reduces overall storage utilization by 43.4%, and results in speeding up the studied jobs by as much as 21.3%.

Acknowledgment

This work was sponsored in part by the NSF under Grant No: CNS-1016408, CNS-1016793, CCF-0746832, and CNS-1016198.

References

[1] Hadoop and Disparate Data Stores, 2012. http://www.greenplum.com /blog/topics/hadoop/hadoop-and-disparate-data-stores.
[2] Alex Loddengaard . Cloudera's Basic Hardware Recommendations, 2010. http://blog.cloudera.com /blog/2010/03/clouderas-support-team-shares-some-basic-hardware-recommendations/.
[3] H. Amur, J. Cipar, and V. Gupta. The case for evaluating mapreduce performance using workload suites. In *SOCC*, 2010.
[4] G. Ananthanarayanan and et al. Scarlett: Coping with skewed content popularity in mapreduce clusters. In *EuroSys*, 2011.
[5] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In *HotOS'13*.
[6] Andy Konwinski et.al . X-tracing Hadoop, 2008. Hadoop Summit.
[7] Apache Software Foundation. Hadoop, 2011. http://hadoop.apache.org/core/, accessed on Nov 6, 2011.
[8] T. Beth and et. al. The scads director: scaling a distributed storage system under stringent performance requirements. In *FAST*, 2011.
[9] Y. Chen and et. al. The case for evaluating mapreduce performance using workload suites. In *IEEE MASCOTS*, 2011.
[10] Y. H. et. al. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *ICDE*, 2011.
[11] M. S. et.al. Improving storage system availability with d-graid. In *USENIX FAST*, 2004.
[12] B. Fan and et. al. Diskreduce: Raid for data-intensive scalable computing. In *SOCC*, 2010.
[13] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP '03*, pages 29–43. ACM, 2003.
[14] Hadoop 0.23.0. ViewFs , 2011. http://hadoop.apache.org/docs/r0.23.0 /api/org/apache-/hadoop/fs/viewfs/ViewFs.html.
[15] S. Huang and et. al. The case for evaluating mapreduce performance using workload suites. In *ICDEW*, 2010.
[16] R. Kaushik and et al. GreenHDFS: towards an energy-conserving storage-efficient, hybrid hadoop compute cluster. In *USENIX ATC*, 2010.
[17] W. Lang and J. M. Patel. Energy management for mapreduce clusters. In *Proceedings of the VLDB Endowment*, 2010.
[18] J. Leverich and C. Kozyrakis. On the energy (in)efficiency of hadoop clusters. In *ACM SIGOPS Operating Systems Review*, 2010.
[19] Z. Li, K. M. Greenan, A. W. Leung, and E. Zado. Power consumption in enterprise-scale backup storage systems. In *USENIX Fast*, 2012.
[20] C. liu et. al. R-admad: High reliability provision for large-scale de-duplication archival storage systems. In *ICS*, 2009.
[21] M. Mihailescu and et al. Mixapart: decoupled analytics for shared storage systems. In *USENIX FAST*, 2012.
[22] M. Poess and et al. Energy cost, the key challenge of data centers: A power consumption analysis of tpc-c results. In *PVLDB*, 2008.
[23] G. Porter. Decoupling storage and computation in hadoop with super-datanodes. *SIGOPS Oper. Syst. Rev.*, 44:41–46, April 2010.
[24] S. Rao and et al. Hotrod: Managing grid storage with on-demand replication. Technical Report YL-2012-004, Yahoo, Apr 2012.
[25] Sameer Tiwari. Managing Hot and Cold Data Using USS, 2012. http://www.greenplum.com/blog/topics/data-warehousing/managing-hot-and-cold-data-using-a-unified-storage-system.
[26] R. Schmidt. Intelligent block placement policy to decrease probability of block loss, 2012. https://issues.apache.org/jira/browse/HDFS-1094.

[27] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *26th IEEE Symposium on Massive Storage Systems and Technologiese*, 2010.

[28] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: helping disk arrays sleep through the winter. In *Proc. SOSP*, 2005.