DiskReduce: Replication as a Prelude to Erasure Coding in Data-Intensive Scalable Computing

Bin Fan   Wittawat Tantisiriroj   Lin Xiao   Garth Gibson

{binfan , wtantisi , lxiao , garth.gibson} @ cs.cmu.edu

CMU-PDL-11-112
October 2011

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Acknowledgements: We would like to thank several people who made significant contributions. Robert Chansler, Raj Merchia and Hong Tang from Yahoo! provide various help. Dhruba Borthakur from Facebook provides statistics and feedback. Bin Fu and Brendan Meeder gave us their scientific applications and data-sets for experimental evaluation. The work in this paper is based on research supported in part by the Betty and Gordon Moore Foundation, by the Department of Energy, under award number DE-FC02-06ER25767, by the Los Alamos National Laboratory, by the Qatar National Research Fund, under award number NPRP 09-1116-1-172, under contract number 54515-001-07, by the National Science Foundation under awards CCF-1019104, CCF-0964474, OCI-0852543, CNS-0546551 and SCI-0430781, and by Google and Yahoo! research awards. We also thank the members and companies of the PDL Consortium (including APC, EMC, Facebook, Google, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, NEC, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, and VMware) for their interest, insights, feedback, and support.

Keywords: Distributed File System, RAID, Cloud Computing, DISC

Abstract

The first generation of Data-Intensive Scalable Computing file systems such as Google File System and Hadoop Distributed File System employed n *(*$n \geq 3$*) replications for high data reliability, therefore delivering users only about* $1/n$ *of the total storage capacity of the raw disks. This paper presents DiskReduce, a framework integrating RAID into these replicated storage systems to significantly reduce the storage capacity overhead, for example, from* 200% *to* 25% *when triplicated data is dynamically replaced with RAID sets (e.g.* $8 + 2$ *RAID 6 encoding). Based on traces collected from Yahoo!, Facebook and Opencloud cluster, we analyze (1) the capacity effectiveness of simple and not so simple strategies for grouping data blocks into RAID sets; (2) implication of reducing the number of data copies on read performance and how to overcome the degradation; and (3) different heuristics to mitigate "small write penalties". Finally, we introduce an implementation of our framework that has been built and submitted into the Apache Hadoop project.*

1 Introduction

File systems for Data-Intensive Scalable Computing (DISC), such as Google File System (GFS) [GGL03], Hadoop File System (HDFS) [Bor09b], or S3 [S3] are popular for Internet services in today's cloud computing. To achieve high reliability and availability with high concurrent disk failures, these file systems replicate data — typically three copies of each file. Thus they suffer from 200% storage overhead, and often discourage or disallow concurrent or sequential write sharing. Alternatively, operating at a comparable or larger scale with more demanding sharing semantics, High Performance Computing (HPC) file systems, such as Lustre [Lus], GPFS [SH02], PVFS [CWBLRT00], or PanFS [WUA$^+$08], achieve tolerance for disk failures with much lower capacity overhead (e.g., 25% or less) by erasure codes or RAID [PGK88] schemes such as RAID-DP [BFK06], Reed Solomon P+Q [Pla97] or EVENODD [BBBM95].

The goal of this work is *to reduce the storage overhead of DISC file systems by erasure coding.* Yet adding erasure coding to DISC file systems is not the same as simply switching to HPC file systems due to at least the following reasons:

- DISC file systems were often designed to follow the model of moving code to data and thus use a large number of commodity hardware serving as both computation node and storage node, rather than using specialized storage clusters equipped with expensive RAID hardware.
- Replication in DISC file systems can be valuable for more than just reliability.
- DISC files are significantly larger than files in traditional systems, even HPC file systems, but the data blocks are also significantly larger (e.g., 64 MB in HDFS and GFS) than blocks in other file systems.

In this paper, we focus on analyzing different design choices associated with building RAID on top of a DISC file systems to balance storage overhead, performance and reliability. Our contributions include:

- We gathered usage data from large HDFS DISC systems and find that DISC files are huge relative to traditional and HPC file systems, but because DISC blocks are also huge, per-file RAID still wastes significant capacity.
- We measured the performance implication of reading RAIDed data for MapReduce jobs . We find that triplicated files can be read at higher bandwidth than single-copy files, as expected, but that this advantage is perhaps smaller than expected, and is absent in many cases.
- As RAID sets formed with blocks from multiple files introduces RAID "small writes" even with immutable files because file deletion must update the erasure code to reclaim deleted space, we investigate different policies to reduce the RAID maintenance cost.

In addition, we also present *DiskReduce*—our implementation of RAID erasure encoding and decoding framework extending HDFS to replicate files initially and trigger background processing to encode triplicated files into erasure coded RAID sets. The implementation of RAID 6 encoding has been submitted to the HDFS Apache project [Bor09b] and is available to all [Tan10].

2 Related Work

GFS, HDFS, HDFS implementation The Google file system (GFS) [GGL03] and Hadoop distributed file system (HDFS) [Bor09b] which is modeled after GFS, defined data-intensive file systems. They provide reliable storage and access to large scale data by parallel applications, typically through the Map/Reduce programming framework [DG04]. Our implementation is a modification on top of HDFS because it is open source. HDFS supports write-once-read-many semantics on files. Each HDFS cluster consists of a single metadata node and a usually large number of datanodes. The metadata node manages the namespace, file layout information and permissions. To handle failures, HDFS replicates data three times.

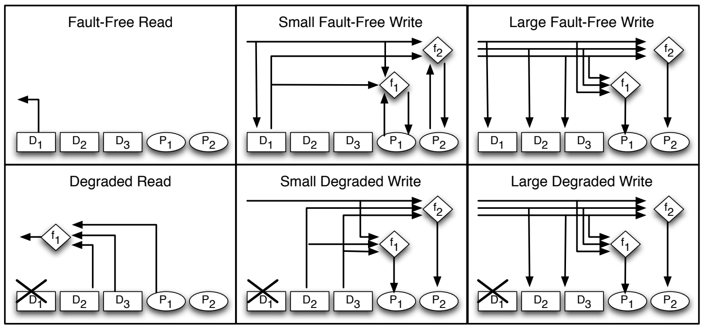
Core RAID Almost all enterprise and high performance computing storage systems protect data against disk failures using a variant of the erasure protecting scheme known as RAID [PGK88]. Presented originally as a single disk failure tolerant scheme, RAID was soon enhanced by various double disk failure tolerance encodings, collectively known as RAID 6, including two-dimensional parity [GHK$^+$89], P+Q Reed Solomon codes [RS60, CLG$^+$94], XOR-based EVENODD [BBBM95], and NetApp's variant Row-Diagonal Parity [CEG$^+$04]. Lately research is turned to greater reliability through codes that protect more, but not all, sets of larger than two disk failures [Haf05], and the careful evaluation of the tradeoffs between codes and their implementations [PLS$^+$09].

Network RAID Networked RAID has also been explored, initially as a block storage scheme [LMC94], then later for symmetric multi-server logs [HO93], Redundant Arrays of Independent Nodes [BFL$^+$01], peer-to-peer file systems [WK02] and is in use today in the PanFS supercomputer storage clusters [WUA$^+$08]. Our system explores similar techniques, specialized to the characteristics of large-scale data-intensive distributed file systems.

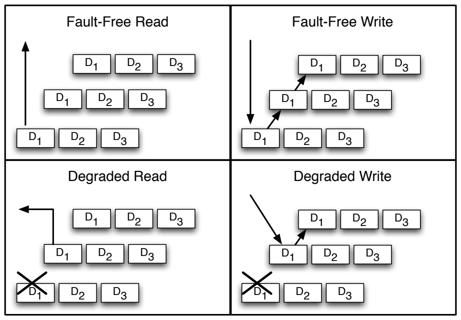
Cache design representation File caching has been widely used in distributed systems to improve performance [HKM$^+$88, NWO88]. The file system [CG91] combines file caching and compression in two levels: one sector of the disk holds uncompressed data which can be accessed at normal disk speed and a sector holds compressed data which needs to be uncompressed before access. The least recently used files are automatically compressed. AutoRAID [WGSS96] is proposed to implement a two-level storage hierarchy in RAID controllers where active data is mirrored and inactive data is stored with RAID 5 protection. Migration of data between two levels is performed in the background based on the least-recently-written data dynamically determined. Our system applies the principle of file caching to exploit temporal locality of data access to balance performance requirement and storage limit given temporal locality in data access. Having the similar two-layer representation of data and background migration, however our system is replicating data among servers instead of using hardware RAID and optimized for the unique usage of cloud data.

File system statistics In this work, design choices are made based on statistics (e.g. file size distribution, file access pattern) collected from clusters for cloud computing. As a comparison, the file size distribution in supercomputing file systems are reported in [Day08]. The access pattern and deletion pattern of UNIX BSD 4.2 file system is reported by a trace study in mid 80s [OCH$^+$85]: most of the file accessed are open only a short time and accessed sequentially; most new information is deleted or overwritten within a few minutes of its creation.

RAID Consistency Previous works address how to reduce the cost of maintaining RAID consistency (e.g. parity), without compromising data reliability. AFRAID [SW96] always applies data update in real time but shifting parity update to idle cycles and therefore eliminates the small-update penalty by hiding the cost, with slight loss of data availability. Paritypoint [CK93] claims it is important for parallel file systems being able to turn off parity on a per-file basis so that applications can disable parity update for temporary files and increase I/O independence. Data is immutable in our system so RAID set is only updated after deletion. The check blocks are



(a) RAID: normal read, normal write (to read either data blocks not being written or all check blocks, whichever is less work, and compute new check blocks as either the full code or the delta on the old code) , single-failure degraded read (read all other data block and at least one check block to decode missing block), single-failure degraded write (read data blocks not being written and compute new check blocks before writing non-failed check and data blocks)



(b) Triplication: normal read, normal write (to three copies), single-failure degraded read (read one of the remaining blocks), and single-failure degraded write (update remaining blocks)

Figure 1. Basic operations of a RAID set with $k = 3, m = 2$ and $k = 1, m = 2$. Arrows represent data flow. Replication, although more expensive in work done and capacity overhead, is simpler than RAID.

updated asynchronously like AFRAID, but deleted blocks are marked in real time. Therefore the consistency is always guaranteed.

Cleaning In our system after deleting blocks in a RAID set, the space has to be reclaimed by garbage collection. This is similar to Log-structured file systems [RO91] where different heuristics are studied to gather the freed segments into clean segments with small overhead [BHS95].

Erasure Coding in Cloud Adding erasure coding to data-intensive distributed file systems has been introduced into the Google File System [Col]. An very early version of this work using RAID5 with mirror to protect double failures was adapted into the Hadoop Distributed File System [Bor09a]. An experimental study [ZDM+10] explores the workload dependent tradeoffs of erasure coding versus replication in cloud data centers. In this paper, we focuses more on analyzing the design choices to balance performance, manageability and storage advantage.

3 Background

RAID Basics RAID [PGK88] is widely used to increase storage reliability through redundancy. In this paper, a *RAID set*[1] of code (k, m) denotes a set of k *data blocks* $D_1, \ldots D_k$ and m *check blocks* $P_1, \ldots P_m$. The code (k, m) is designed to recover all data after m different block failures, which requires

1. The check blocks P_i are each calculated from all data blocks $\{D_1, \ldots, D_m\}$ and are *independent of each other*. In practice, the first check block P_1 is usually a parity block, i.e., the bitwise exclusive-or (XOR) of all data blocks and codes with $(k > 1, m = 1)$ are known as RAID5(or more rarely as RAID3 or 4)[PGK88]. To tolerate double failures, P_2 can be calculated using XOR with coding schemes[Pla97] such as EVENODD [BBBM95], RAID-DP [BFK06], or using

[1]also referred to as a RAID stripe or RAID group in previous literature.

polynomials over a Finite Field using a Reed-Solomon code [RS60]. Common usage calls all of these ($k > 1, m = 2$) codes RAID6.

2. Blocks in the same RAID set must be *assigned to different failure domains*[2]. Traditional RAID systems always map the blocks of one RAID set to a fixed set of disks in the same pattern and repeat for all RAID sets. However, in Panasas's PanFS[WUA+08], GFS, HDFS and DiskReduce, the storage nodes associated with each RAID set are chosen independently, a scheme for load balancing RAID work known as declustered RAID [HG92, ML90].

RAID v.s. Replication Replication is also widely used to provide data reliability. In fact replication can be viewed as a degenerate RAID code with $k = 1$. Figure 1 compares the operations involved in reading and writing data with RAID 6 ($k = 3, m = 2$) and triplication ($k = 1, m = 2$). The use of replication versus RAID is a multi-way tradeoffs:

- Replication is simpler.
- Replication does less work except for large write intensive workloads.
- Replication may get performance advantage due to higher disk bandwidth from more spindles, and better chance to balance the load of disks.
- Replication consumes significantly more capacity.

HPC and DISC File Systems The two different architectures widely used for large-scale storage clusters are:

- High Performance Computing (HPC) file systems, typified by Lustre [Lus], PanFS[WUA+08],GPFS [SH02], or PVFS [CWBLRT00]. HPC usually separates data storage nodes from compute nodes and uses RAID controller pairs to provide reliable data access.
- Data Intensive Scalable Computing (DISC) or Cloud Computing storage, represented by GFS and HDFS. Each node in DISC not only provides computation but also stores the data. To handle node failures in DISC without relatively expensive externally RAID storage, data is replicated (triplication by default) across different nodes.

There are exceptions to this taxonomy. For example, Amazon EC2, EB3 and S3 [S3] separate storage services and compute services into different nodes but still use replications among storage servers nodes. In HPC, Panasas PanFS [WUA+08] separates nodes for storage service but uses software RAID across nodes instead of embedded RAID controllers.

Our goal is to enhance the DISC storage model because it pays too large a storage overhead (e.g., 200% for triplicating data) with software RAID techniques to lower the overhead (e.g., to less than 25%), without sacrificing failure protection.

HDFS Basics Our work is based upon Hadoop Distributed File System (HDFS)[Bor09b]. HDFS is an open-source data intensive file system widely used in modern DISC clusters. In design it is very similar to the Google File System (GFS)[SRO96]. Each HDFS cluster consists of a centralized metadata server process called *namenode* and a large number of data server processes called

[2]Here we assume the fault model consists of both independent and dependent failures such as disk failure, node failure, rack failure, and RAID sets are selected so that each dependent failure will cause at most one loss per RAID set. Failures such as loss of a whole data center are outside of the scope of this paper.

datanodes. The namenode process manages the namespace, file layout information and permission control. Each datanode process manages data on local storage and serves data to HDFS clients. HDFS is designed and optimized to provide high throughput and high availability for very large data sets and differs from traditional file systems in the following ways:

- Files have a single-writer, write-once-read-many semantics and are immutable once closed.
- Namenode keeps all metadata in memory to achieve high performance.
- Files are divided into "very large" blocks of 64 to 128 MB (compared with 4KB in traditional file systems) to (1) facilitate data streaming for applications such as MapReduce [DG04] and to (2) avoid overloading namenode with too much metadata or too frequent changes to metadata.
- Each data block is replicated across multiple datanodes — typically two in the same rack and one in another rack, thus data is still accessible with two node failures or one rack (e.g. switch) failure. The side effect of replication is a storage overhead of 200%.

4 Challenges

The basic idea of building RAID using (k, m) code on top of HDFS is straightforward: group k different HDFS blocks into a RAID set, and then generate m check blocks to protect these k blocks. All $k + m$ blocks of this RAID set are stored on $k + m$ different datanodes to survive from m simultaneous node failures, and with a background re-replication process used to recover the missing data. However due to the unique design features of HDFS as described in Section 3, there are several challenges:

How to achieve low storage overhead in real HDFS environment. Since each HDFS block is large (64 MB by default), if we follow the common rule used by lots of traditional RAID systems that only groups blocks from the same file into the same RAID set[3], it may not be trivial to find enough number of blocks into one group and having insufficient blocks in the RAID set leads to higher capacity overhead.

One way to solve this problem is to decrease the size of HDFS blocks. But then the increased amount of metadata (e.g. mapping from block to the datanodes serving it) is likely to overload the namenode which keeps metadata in memory. Another way is to divide blocks into sub-blocks to make RAID sets and use deterministic algorithms to compute sub-block locations. Nevertheless, this will significantly restrict the flexibility of HDFS to handle adding or losing servers. We study the cost of different grouping strategies using file statistics from real HDFS clusters in Section 5. Our implementation still uses 64 MB blocks, but allows blocks from different files to make one RAID set.

How performance degrades with fewer number of copies. After encoded into RAID 6, the number of clear copies of each data block is reduced from three to only one. We explore the performance degradation for reading data by MapReduce applications that might result from having fewer sources in Section 6.

Our implementation, in order to hide the potential performance degradation, makes encoding asynchronously. In other words, Our system initially triplicates data the same as HDFS. Encoding is triggered and performed in the background. By delaying RAID encoding, a (MapReduce) task scheduler has more options to exploit collocation of computation in a node containing the needed

[3]one major advantage of making per-file RAID is to make deletion easy

data. In Section 5, we also discuss the opportunity to apply a cache hierarchy which treats the triplicated data as in cache as to improve the performance.

How to mitigate the small write problem. Traditional RAID systems suffer from performance degradation when updating data smaller than a single RAID set, due to the read-modify-write sequence to update the check blocks. Although files in HDFS are immutable, the small write problem could still happen if a RAID set contains blocks from different files and one of the files is deleted. To keep RAID consistent (i.e. the remaining blocks recoverable) , we need to either pay extra work for new check blocks, or to mark the the "deleted" blocks invisible from users. In Section 7, we explore different strategies to mitigate the small write problem.

How to reduce the deployment/implementation complexity. When we talk to HDFS developers, they are reluctant to use RAID because they are concerned about error handling on the critical path. In fact this is consistent with observations from Panasas when dealing with error under concurrent access and error cases. Since RAID encoding in our system is an asynchronous process decoupled from the critical path, background encoding can be deferred if error are encountered which makes the error handling easier. By encoding data asynchronously, our change can be isolated from some or all of the HDFS nodes. Deployment thus becomes easier. In Section 8, we introduce our prototype implementation of our framework as an external tool to HDFS.

In the following sections, we explore the tradeoffs between storage overhead, performance, data reliability to address these challenges. We evaluate our experiments and provide most of the analysis based on RAID 6 encoding ($m = 2$). However our framework is independent of the RAID code and any RAID code can be applied to provide protection of different levels.

5 Encoding

Reducing the storage capacity used by redundant information is the most important reason for using RAID in HDFS. However its effectiveness may be limited by the large block size used in HDFS. In this section, we compare two strategies for grouping blocks into RAID sets.

Per-file RAID is a strategy that restricts one RAID set to the blocks *from the same file.* This strategy is simple and has been adopted in storage systems including HDFS-RAID [Bor09a] and PanFS [WUA$^+$08]. RAID per file ensures no permission conflicts accessing a RAID set and it facilitates encoding in the writing node before data propagates to data nodes. But the major benefit of this strategy is to ensure that whenever a block is deleted, because its file is deleted, all the other blocks in this RAID set are also deleted. Consequently deletion will not require re-encoding. However, RAID per file applied to "small" files (e.g., a file of size less than 128MB may only contain 2 data blocks) has large (100%) storage overhead.

Across-files RAID is a strategy allowing blocks from different files to be grouped in the same RAID set, ensuring all RAID sets are as large as desired and storage overhead is correspondingly low. However in addition to access control issues, adding or deleting a block in a multi-file RAID set requires recomputing check blocks, as shown in Figure 1(a) and known as the *small write problem.*

While RAID per file has compelling advantages, we suspect that it gives up significant capacity with small RAID sets. For example, HDFS-RAID [Bor09a] does not encode any file with less than two blocks (e.g., 128MB). In HPC systems, PanFS uses a 64KB block size to effectively eliminate this capacity overhead, but its metadata is not limited to the memory of one machine and it does not try to locate 64MB at a time in just one machine. In the rest of this section we show the capacity benefit possible with RAID across files and leave the small write problem to Section 7.

File Size Distribution

ID	Description	# Nodes	Raw Capacity	# Files
DW	unknown	2000	21 PB	40.0 M
OC	Research	64	0.3 PB	5.7 M
M45	Research	400	1.5 PB	21.8 M
A	Sandbox	1800	3.8 PB	41.8 M
C	Research	800	3.5 PB	10.1 M
D	Production	3000	6.5 PB	23.4 M
M	Research	2000	6.3 PB	25.5 M
N	Research	3500	10.6 PB	35.4 M
U	Production	1700	13.0 PB	17.7 M

Table 1. Clusters from which we collect file size distributions. DW (Data Warehouse) is at Facebook, OC (Opencloud) at CMU, and all other clusters are at Yahoo!. DW is the largest capacity HDFS cluster we know about as of May 2010 [Fac].

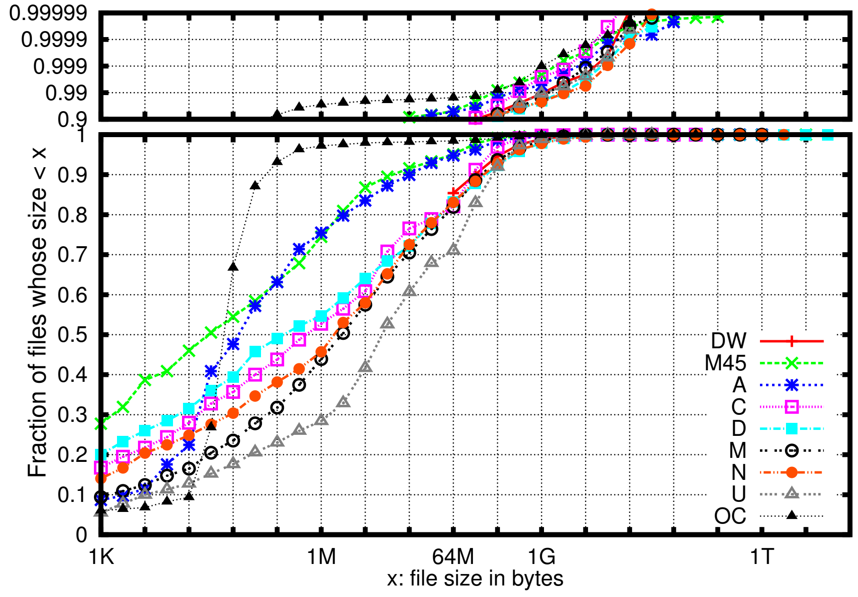
To compare these two strategies in real HDFS clusters, we gathered file size data from HDFS clusters in CMU, Facebook and Yahoo! (summarized in Table 1). Figure 2(a) plots the cumulative distribution function (CDF) of file size. Across all clusters, the largest single file observed is about 2.7 TB; the median size ranges widely from 16 KB to 6 MB, and the average ranges from 8 MB to 108 MB. Compared with HPC file systems reported by Dayal et al [Day08] where median file size ranges from 2 KB to 256 KB and mean from 169 KB to 29 MB, file systems for cloud computing have considerably bigger files. Yet even with these "large" files, at least 70% and as many as 99% of files are smaller than 64MB (1 HDFS block), as shown in Figure 2(a).

Common wisdom says that small files are numerous but take little total space so the capacity overhead for small files might be negligible. Figure 2(b) plots the CDF of space used by files up to a particular size. Across all clusters 40% to 80% of the storage capacity is consumed by files smaller than 1GB (16 blocks); in cluster C, at the extreme, 55% of capacity is used by files smaller than 256 MB (4 blocks). Facebook's DW cluster, the largest, has some 2PB of files smaller than 256MB. Because four block files will often have poor RAID overhead, we expect RAID across files to be more efficient.

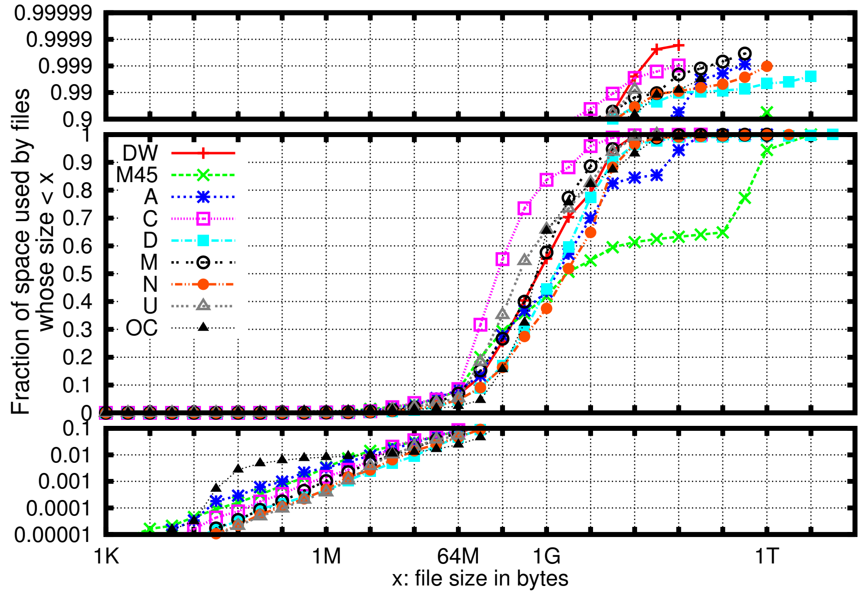
Figure 3 shows our calculation of the storage overhead that would be induced by RAID in these clusters. In Figure 3 we construct across-files RAID sets from all the blocks of the files in the same directory, RAID per directory. This is a sample to implement approximation of RAID across any file. Generally, RAID per file leads to a higher capacity overhead due to the large number of "small" files. For example, averaged on all clusters, the ($k = 4, m = 2$) storage overhead is about 75%, while if all RAID sets were full the overhead would be $2/4 = 50\%$. When k is larger, lowering the best case overhead, the gap between the average RAID per file overhead and the optimal RAID 6 overhead is still larger. Ideally the ($k = 16, m = 2$) code has an overhead of 12.5% but RAID per file only achieves an average overhead of about 50%.

In contrast, RAID per directory overhead performs very close to the ideal RAID 6 overhead across all clusters even when k is small. Essentially, directories are large enough to construct full RAID sets with big blocks.

Because the storage overhead for check blocks in RAID per directory is, roughly, half what it is with RAID per file, the rest of this paper focuses on RAID across files instead of RAID per file. Of course even RAID per file has a storage overhead that is much less than triplication, and it is simple to implement, so we are not surprised that it is used in today's HDFS-RAID [Bor09a].



(a) Fraction of files $\leq x$ bytes



(b) Fraction of storage used by files $\leq x$ bytes

Figure 2. File statistics (cumulative distribution function, CDF) collected from HDFS clusters listed in Table 1 (using the cluster IDs in that table). Figure (a) shows the CDF of file size and (b) shows the CDF of the total capacity used in files up to a given size.

6 Read Performance

Triplicated data is available from three disks in three nodes, increasing the total disk bandwidth available for reading that data. This has led to more than three replicas for performance rather than reliability [SRO96]. In this section we study how reducing the number of data copies impacts

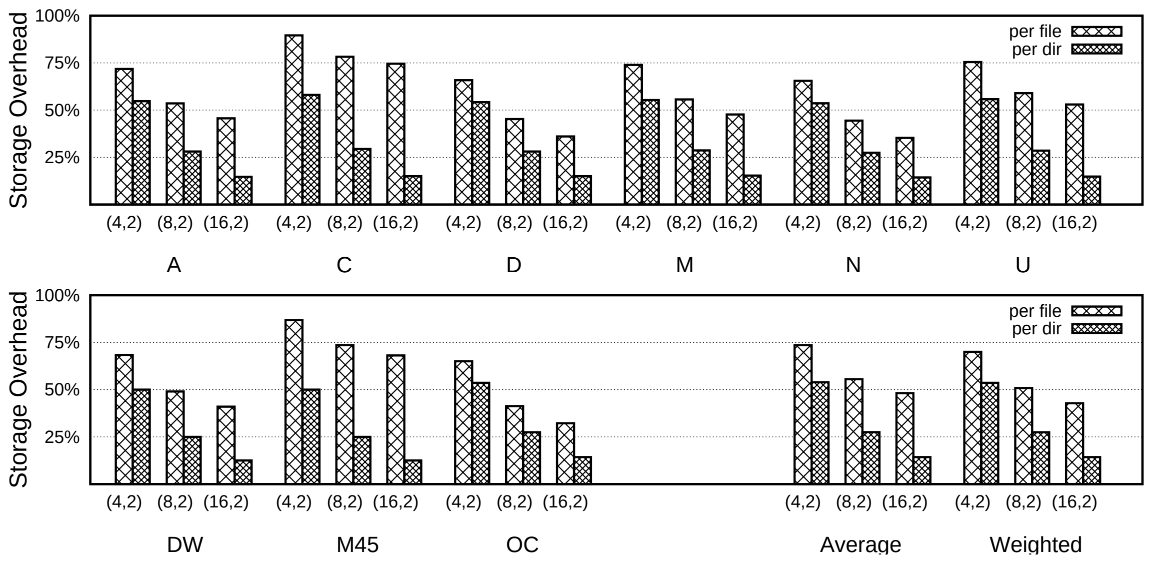


Figure 3. Storage overhead for check blocks with RAID sets restricted to blocks of the same file and directory for the clusters in Table 1, with different RAID set sizes: k = 4, 8, 16 and m = 2. We also show the average overhead treating each cluster equally and weighting each cluster by its total capacity.

read performance. Then we investigate how to overcome the potential performance degradation by exploiting temporal locality in data access.

To study the impact on read performance when data has different replication levels, we tested a private HDFS cluster with one namenode and 50 datanodes and running Hadoop in the same nodes. Each node has two quad-core Xeon processors of 2.66GHz, 16 GB of memory, one 1TB 7200-rpm SATA disk and a 1 Gigabit Ethernet.

Our test uses a MapReduce job to output 1200 blocks (75 GB), configured with N reducers and with R replications. Parameter N varies from 10 to 50 and R is 1 or 3. Because in Hadoop each reducer creates one file containing all output, N reducers will write N different files. Since the first copy of any block is written locally, if $R = 1$ all blocks in an output file are stored in the reducer node. As a result, if only 10 reducers generate all 1200 blocks with only 1 copy, these 1200 blocks will be distributed only on 10 datanodes, each having 120 blocks. With 3 copies, while 10 datanodes will each have a complete local copy of one output file, the remaining two copies of any block will be spread randomly across the entire cluster. This microbenchmark gives us a convenient way to generate evenly distributed (N=50) and unevenly distributed (N=10) data.

Figure 4 shows the completion time when all 1200 blocks are read by 1200 maps in a map-only Hadoop job, as the average of 13 runs. When all 1200 blocks are replicated three times the read time is insensitive to the number of files that partition the blocks. Figure 5 shows that Hadoop is able to find a datanode containing the needed block for almost all maps when the data is triplicated.

However, when there is only one copy of each block, the time to read all blocks is sensitive to how the files were written – the time to read 10 files with one copy is 6 times longer than the time to read the same data written into 50 files. Figure 5 shows that with one copy Hadoop is not able to find a node with a copy 40% of the time.

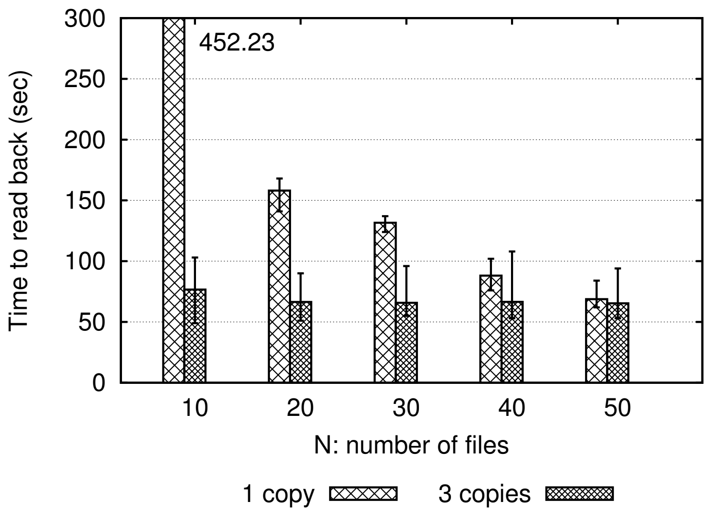


Figure 4. Time to read in a "hotspot" benchmark

ID	Duration	HDFS Log type
M45	590 days	Namenode/Datanode Log
DW	30 days	Namenode/Datanode Log
A	440 days	Namenode/Datanode Log

Table 2. Logs with access time details.

Our point is that there are certainly cases where a RAID encoding, having only one copy of each data block, could have slower read performance than when all data is triplicated. [4]

6.1 Temporary Locality and Caching Hierarchy

Caching is widely used to temporarily represent data in a faster location. To protect against potential performance degradation of reading the RAID encoded data, we propose a triplicated cache of recently written data. Namely, the storage of data is divided into a triplicated cache layer providing better performance and a RAIDed layer saving space capacity.

To predict the effectiveness of such a cache, we gathered detailed logs from Facebook and Yahoo! clusters, described in Table 2. Based on these logs, we compose the creation and read access time of all data blocks. For each read, we calculate the "block age". The cumulative distribution function (CDF) of this block age on read accesses is shown in Figure 6. Over 90% of block accesses in the M45 and DW clusters happen within the first day after creation and in the A cluster, more than 50% of block accesses occur in the first day.

The temporal locality shown in Figure 6 suggests that a triplicated data cache is likely to satisfy most block accesses in many clusters, delivering whatever performance benefit if any triplication has to offer. When data is newly generated, it is triplicated until it falls out of the cache and is RAID encoded to save space.

[4]Interestingly, the 50 file case in Figure 4 suggests that large data sets, written and read with full parallelism, will often see no degradation in read bandwidth when RAID encoded.

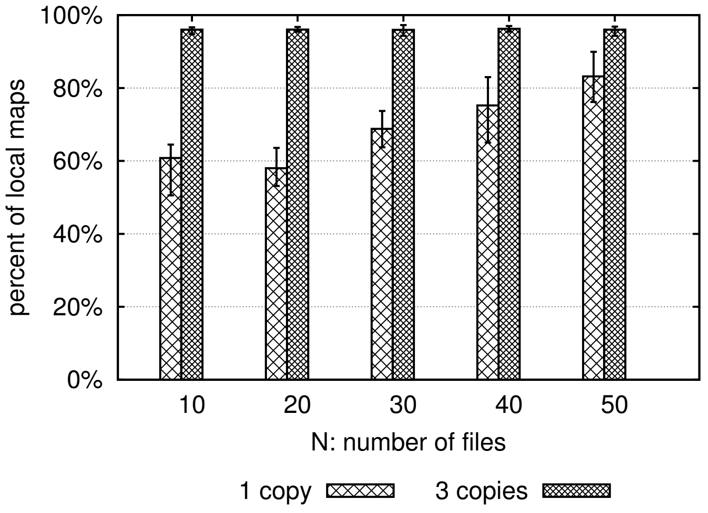


Figure 5. Fraction of map tasks that are local

Number of user blocks to cache						
1K	2K	4K	8K	16K	32K	64K
17%	20%	23%	31%	45%	65%	81%

Table 3. Cache hit ratio, LRU replacement policy

Table 3 shows an LRU caching simulation of the M45 cluster trace over 564 days. In this trace more than 60 million user blocks are created, on average more than 100,000 per day. Table 3 suggests that, if we keep different blocks triplicated, the cache hit ratio would be 80%. In terms of cost, 64,000 triplicated blocks are less than the average number created in one day and use less than 10% of the capacity of M45.

7 Deletion

Traditional RAID systems suffer a performance degradation when updating data smaller than one RAID set, compared to updating a whole RAID set, known as *small write problem*[PGK88], because they have to execute a *read-modify-write* sequence to update the check blocks in addition to the data blocks. In HDFS files are immutable, yet this small write problem will still occur when deletion happens in RAID sets built from multiple files. For example, if we want to delete one block from a RAID set and recover the capacity of that block, the corresponding check blocks have to be recalculated without it in order to ensure that the RAID equation is still consistent and the blocks remaining in the RAID set are still recoverable. We will address appending to blocks in Section 9.2.

One simple way to fully eliminate the small write problem in HDFS is to only use per-file RAID restricting all blocks in the RAID set to be from the same file, as is done in "HDFS RAID"[Bor09a] and PanFS [WUA$^+$08]. This simple strategy ensures that any RAID set will always be deleted as a whole. Though simple, Section 5 argues per-file RAID leads to a storage overhead much higher than RAID across files.

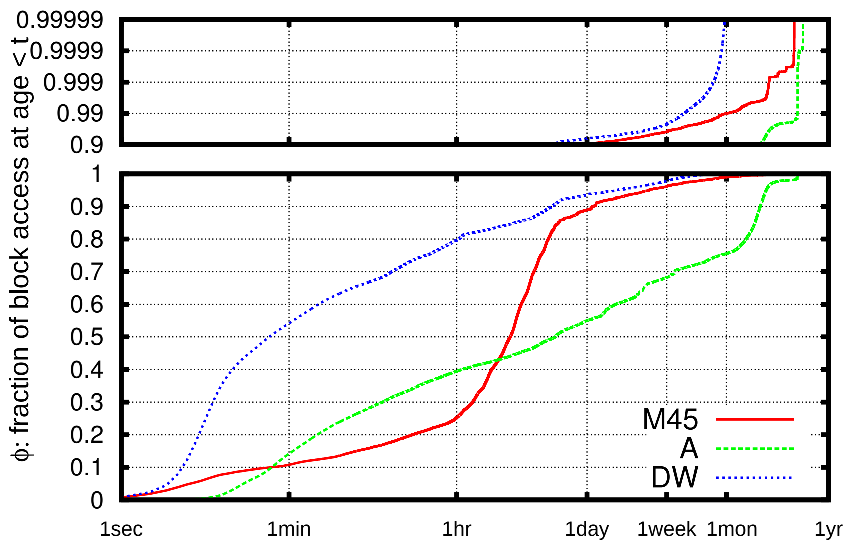


Figure 6. Distribution of block age at read access time.

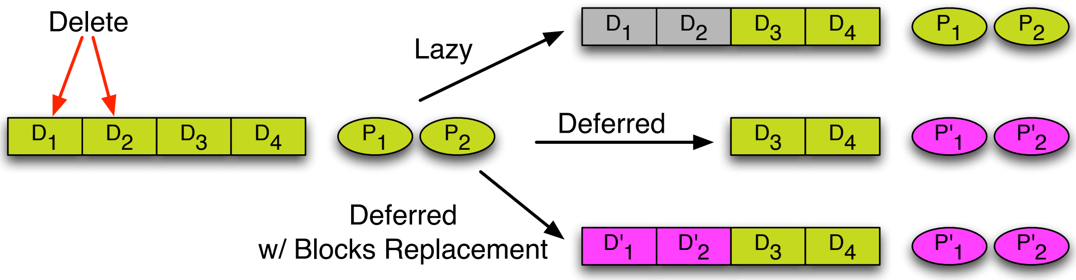


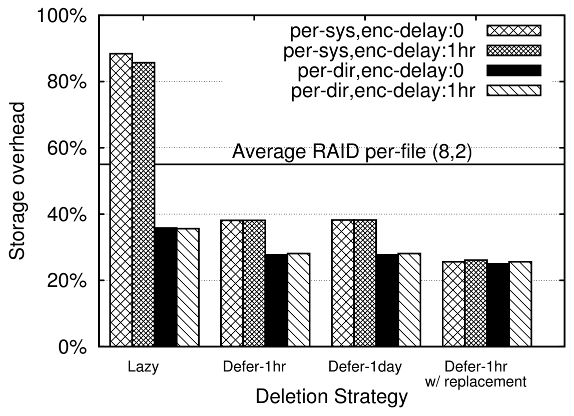
Figure 7. Three different ways to handle deletion

In this section we investigate three strategies illustrated in Figure 7 to handle deletion when across-file RAID is implemented:

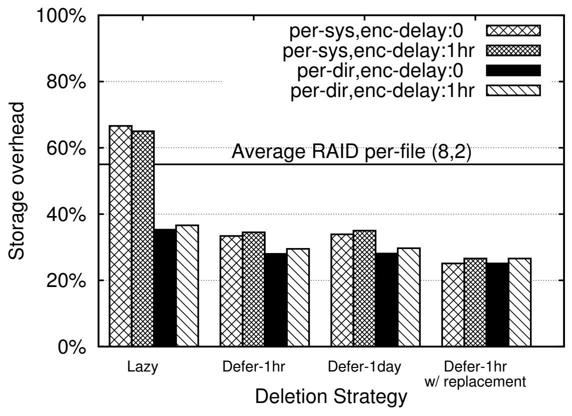
1. Lazy Deletion never recalculates the RAID equation but instead marks deleted blocks "unseen" until all blocks in a RAID set are deleted, then it recovers the space of all blocks. Lost or failed blocks can still be recovered from the old RAID equation. However, the space in deleted blocks can not be reclaimed until all blocks in this RAID set are deleted. This strategy pays zero cost for maintaining RAID consistency on deletion, but may cause capacity to "leak" into partially unseen RAID set.

2. Deferred Deletion does not recalculate check blocks immediately after each block deletion. Instead, it delays the recalculation in case the entire RAID set is deleted before a timeout. This scheme exploits temporal locality in deletion as it is likely that one deletion is soon followed by related deletion. All unseen space will be reclaimed after a delay, and the cost of one "read-modify-write" operation may be amortized over multiple deletes. The weakness of this strategy is that after eliminating deleted blocks from a RAID set, the RAID set becomes "shorter" (i.e. it has fewer data blocks), thus the storage overhead for its check blocks is increased.

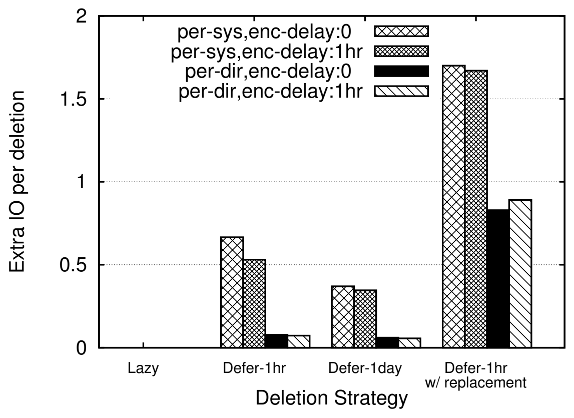
3. Deferred Deletion with Block Replacement also defers recalculation, but when it recalculates check blocks it also adds new blocks into the existing RAID set to prevent the RAID set from becoming shorter. This approach is similar to the small write process in traditional RAID systems where the deferred recalculation is provided by a non-volatile write back cache in the RAID controller [MC93]. One weakness of this strategy is that it is likely to make the blocks in a RAID set more diverse, reducing the chance of a clean delete in the future.



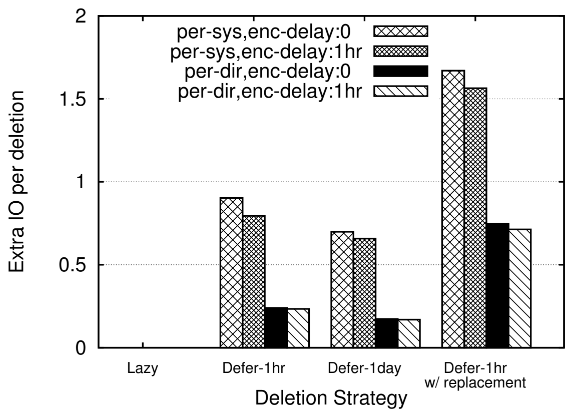
(a) Storage overhead, trace1



(b) Storage overhead, trace2



(c) Storage overhead, trace1



(d) Storage overhead, trace2

Figure 8. RAID across files deletion costs for $(k = 8, m = 2)$ RAID sets. Storage overhead is compared to RAID per file in Figure 4. Extra block IO per block deletion measures the work done in recalculation.

To evaluate these strategies, we run a trace driven simulation using two different traces from Yahoo M45, each containing the creation and deletion events across the entire cluster over 2000 hours (about 80 days). In each simulation, we modeled two different across-files block select policies: per-directory as discussed in section 5 and per system, where RAID sets are selected by namenode from the consecutively created blocks. We consider RAID per system because it is easier to implement online and undelayed in namenode in block creation. Encoding has a configurable delay so if a block is deleted before encoding, it never joins a RAID set. When a block is deleted after joining a RAID set, it is marked “unseen” and the check blocks are not recalculated until configurable re-encode time out occurs. [5]

Figure 8 shows the storage overhead at the end of each 80-day simulation. Because clusters tend to be operated for 3-5 years, an 80 day trace is only 5-10% of the lifetime of the cluster. Accordingly, increases in storage overhead from 25% to 30% in 80 days predicts that the large advantage RAID across files has over RAID per file will be lost in about a year. For this reason we reject all lazy deletion strategies and the deferred deletion strategies using RAID per system. The strategies which appear to sustain an advantage over RAID per file are the block replacement strategies or the RAID per directory with deferred deletion strategies.

Figure 8(c) and 8(d) show the extra block IO done per block deletion. Lazy deletion never recalculates so it does no extra block IO, but the other schemes may have to read partial RAID sets to recalculate check blocks, then write new check blocks. These figures show that the deferred deletion with block replacement strategies pay 3-10X as much extra IO as the deferred deletion strategies.

In summary, with delayed encoding and delayed deletion, RAID per directory sustains its storage overhead advantage over RAID per file. Moreover the RAID per system strategy is not as effective and the block replacement on delayed deletion may do unnecessary extra work.

8 Prototype Implementation

We built a tool and a library illustrated in Figure 9 layered on top of and independent of HDFS. This tool encodes directories into RAID sets and repairs corrupted files, and the associated library can detect and correct missing data while reading. Without comments, the tool and library consist of less than 2,700 lines of Java code, including a RAID 6 erasure code based on the open-source Jerasure coding library [PSS08]. This implementation is available via MAPREDUCE-2036 [Tan10].

In this implementation, encoding and repairing are done asynchronously. The tool invokes a MapReduce job to do encoding and (permanent) repair. While encoding, blocks from different files in the same directory are grouped together and the grouping information is stored as an HDFS file in the directory similar to the Hadoop Archive [HAR] design. Currently, deletion is only supported for whole directories. Before repairing, unmodified HDFS FSCK is used to enumerate the lost of un-replicated blocks. With this list, our repair tool launches a MapReduce job to recompute all damaged files using RAID6 check blocks.

This implementation has a simple deployment strategy. An encoded file can still be read by unmodified HDFS (as an under-replicated file). To benefit from online reconstruction of lost data, an Hadoop application uses our reading library. Online reconstruction is done by catching an exception thrown from the HDFS layer, the technique used by HDFS-RAID [Bor09a].

This layered approach also has limitations. It doesn't have control over HDFS block placement and may need to migrate blocks to respect RAID's requirement for distinct fault domains. Also, to repair a corrupt file, it rewrites the entire file because HDFS files are immutable. Finally, the implementation does not handle append (see Section 9.2).

Experimental Setup We demonstrate our prototype on a cluster of 61 nodes where one node serves as a master while the others serve as clients and slaves for MapReduce and HDFS. Each node has two quad-core 2.83GHz Xeon processors, 16 GB of memory, and four 7200 rpm SATA 1 TB Seagate Barracuda ES.2 disks. Nodes are interconnected by 10 Gigabit Ethernet and Arista

[5]This deletion simulation did not include an LRU cache so encoding is triggered by a timeout rather than eviction. We felt that an explicit timeout is easier to reason about.

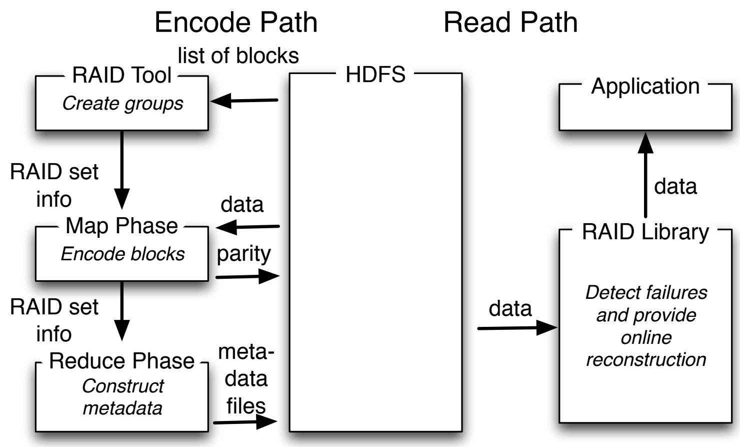


Figure 9. Encode and Read Path for RAID files

10GE switches. All nodes are runing Linux 2.6.32 with XFS as the local file system to store HDFS blocks. Data intensive jobs in this cluster are able to move data to and from disks at about a maximum of 6GB/s.

Methodology We ran the prototype for 30 experiments, each with three phases: writing, encoding and reconstructing. Each experiment constructs a directory that has 3,840 files of size 64 MB each, or 240GB (1GB per disk on average) in total. The size of each file is intentionally set to be one block so that we can estimate the reconstruction speed if it were possible to repair a single block in a larger HDFS file. We employ (8,2) erasure code to encode, i.e. 8 data blocks and 2 check blocks in each RAID set. To trigger reconstruction, we make two nodes offline before running the tool to repair all damaged files. The throughput is measured only when all workers are busy (before 90% of maps complete), and buffer caches are flushed between phases.

Table 4 shows the throughput in terms of user data for each phase and the total disk bandwidth consumed. Since the peak of the disk bandwidth is about 6GB/s in this cluster, writing is near saturation, encoding is 20% less than disk saturation and reconstruction is going at about one third of the disk peak, though we expect this to improve.

Operation	Throughput GB/s (stdev)	Disk I/O GB/s (stdev)
Write (triplication)	1.93 (0.06)	5.80 (0.18)
Encode (RAID6 8+2)	3.69 (0.34)	4.61 (0.43)
Repair (2-node failure)	0.23 (0.02)	2.09 (0.19)

Table 4. Demonstration throughput

Because there are four 1TB disks in each node, with all data triplicated, it will take about 6 hours to encode all 80TB user data to RAID 6 and free up space for another 140TB of new data. Reconstruction is not as efficient but the loss of two full nodes (8TB) will be repaired in less than 10 hours. Large clusters would reconstruct faster because RAID sets are declustered [HG92, ML90] –all nodes participate in reconstructing the same 8TB data as the system gets more nodes.

8.1 Hadoop applications

To demonstrate this prototype in a real use, we run three Hadoop applications. The first two applications were obtained from users of a research cluster while the last one is a generic Hadoop sort application. The first application, *sampling*, is one phase of a distributed astrophysics algorithm, DiscFinder [FRL+10], that identifies large-scale astronomical structures from massive observations and simulation datasets. It reads the input dataset to determine the appropriate partitions while generating a negligible-size output. The second application is a *Twitter analyzer* which processes raw Twitter data into a different format for other tools to operate on [KMF10]. The last one is a *sort* application.

We used the same setup as described earlier. Each application executed three times with four phases: running an application with a triplicated input data-set, encoding the dataset, running the application again with the encoded dataset, and encoding the output dataset. Encoding constructs RAID sets with 8 data blocks and 2 check blocks.

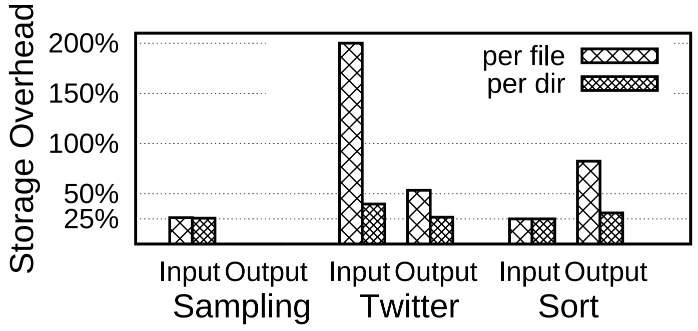


Figure 10. Overhead of RAID per-file and per-dir for each application's input and output

Figure 10 shows the storage overhead of per-file and per-dir RAID 6 (8,2) for the input and output of each application. For the inputs of *sampling* and *sort* application where each file is larger than 1GB, the overhead of RAID per file is small and comparable to the overhead of RAID per-dir. For the input of the *twitter* application where each file is much smaller than 64MB, the overhead of RAID per file is as high as 200%. In the Hadoop framework, each reducer generates one output file. In general, the size of each output file is inversely proportional to the number of reducers. However, most applications choose to use a large number of reducers in order to increase parallelism. For the outputs of the *sampling* and *sort* applications, with the configured number of reducers, the size of each output file is less than 256MB. In this case, the overhead of RAID per file is 54% and 83%, respectively, while the overhead of RAID per-dir is only 27% and 31%, respectively. Using per-dir RAID, an application is flexible to choose a number of reducers to optimize for performance without regard for storage overhead.

Figure 11 shows that the time each application takes when input dataset are triplicated or encoded are comparable. When an input dataset is large enough to spread across all nodes in the cluster, the benefit of an extra copy or two is small. The input size for *sampling*, *twitter*, and *sort* applications are 143 GB, 24 GB, and 120 GB, respectively, and the number of blocks used are 2340, 1504, and 1920, respectively.

Figure 11 and Table 5 show that encoding the output of *twitter* takes 11.3% extra time and 19.5% extra work while encoding the output of *sort* takes 15.6% extra time and 13.1% extra work. For a cluster with these workloads, an application must be 20% slower if encoding is done during

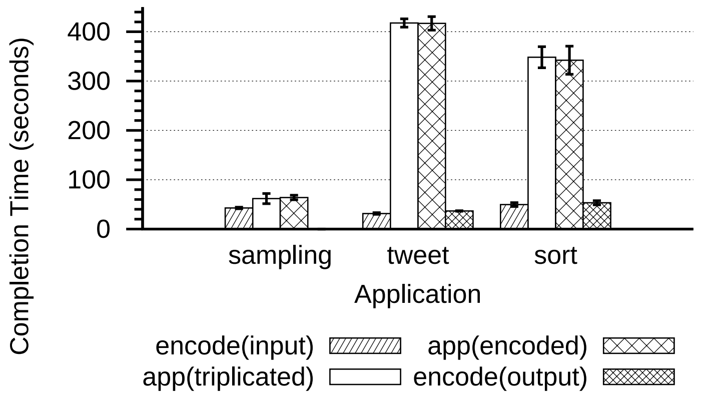


Figure 11. Time to complete each phase

Operation	Read GB	Write GB	Total GB
Twitter (application)	81.1	282.7	363.8
Twitter (encoding)	56.1	15.0	71.1
Sort (application)	544.0	759.7	1303.7
Sort (encoding)	130.8	40.5	171.3

Table 5. Total I/O for running application and encoding its output

busy time. On the other hand, if encoding is done during idle time when it does not slow other work, there must be at least 20% of idle time.

9 Discussion

9.1 Synchronous v.s. Asynchronous encoding

Synchronous RAID encoding creates RAID check blocks immediately on the critical path of writing data The primary advantages of synchronous RAID encoding on HDFS is that it generates less disk and network bandwidth writing data. If a (m, k) code is used, each RAID set containing m data blocks triggers $m + k$ disk writes and $m + k - 1$ network transfer (all blocks have to be on different datanodes with the writer may keep one locally). Thus the amortized cost for adding a block is $1 + k/m$ writes to disk and $1 + (k - 1)/m$ blocks transfer over network, compared to triplication where adding each data block incurs 3 disk writes and 2 network transfer.

Implementing synchronous RAID on HDFS also makes RAID set selection harder. RAID per system is feasible, but inefficient, while RAID per directory is more complex. Moreover synchronous RAID encoding exposes applications to the risk of read performance problems.

In contrast, *Asynchronous RAID encoding* does not perform RAID encoding on the critical path but instead does it in the background. This approach costs more I/O and network resources than triplication or synchronous encoding as it performs triplication first, then reads back and encodes data afterwards. However, it may benefit from deletion from the triplicated cache, or deletion of all blocks in a RAID sets before recalculation. Moreover, in conversation with the HDFS developers, they are reluctant to use RAID because they are concerned about the complexity of error handling on the critical path. By encoding data asynchronously, it can be deferred and retried if any error or complexity is encountered.

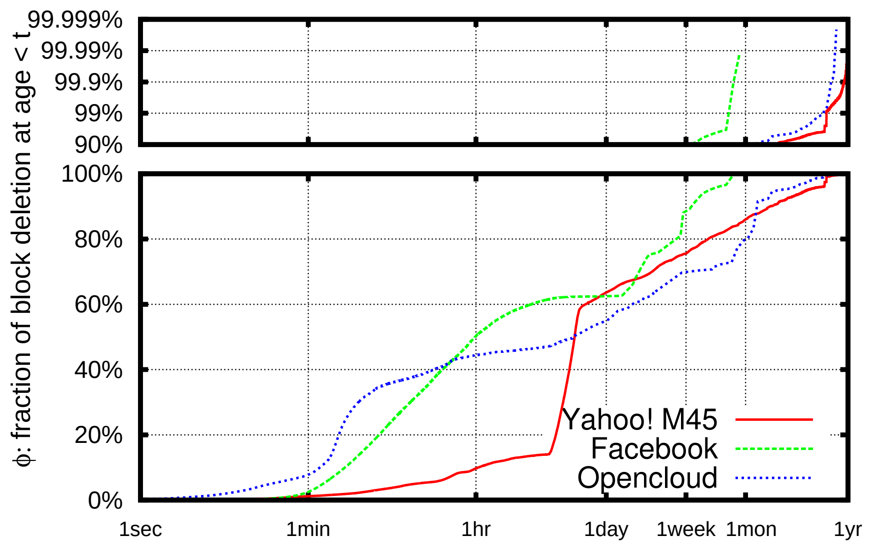


Figure 12. Cumulative distribution of block age on deletion.

9.2 Append

Early versions of HDFS did not support file append operations. Thus a file becomes immutable once closed, and could only be modified by making a new copy with a different name. Since version 0.21.0, HDFS supports file append operation which breaks the write-once semantics.

Though we assume immutable data in HDFS in previous analysis, there are different ways to incorporate RAID into HDFS when append is enabled:

Keep last blocks uncoded: Since the append operation only affects the last block of any file, one simple design is to triplicate the last block instead of encoding it. To append a file, only the last block which is in triplication needs to be mutated. However, this approach may suffer from high storage overhead as we see many files are "small".

Triplicate the block being appended. To achieve low storage overhead, all blocks including the last block of any file will be encoded. If a file has been encoded and the last block of that file is not full, to append the file, the last block is re-created in a triplicated form with data copied from the old version. The old version of the block will be deleted. Then, the data will be appended to the newly created block. Both the deletion and new block creation add extra cost.

Triplicate the data being appended. Instead of triplicating the old block first for every append operation, we can triplicate only the data appended. For append operation, an empty block is created and data is written to this block. The metadata of this file needs to be changed so that the system knows a new block is added to the file. It requires that HDFS allows non-full blocks other than the last block. When the system is idle, these data can be encoded to the RAID set.

10 Conclusion

This paper proposes a framework extending HDFS by initially replicating blocks, then converting them into RAID as they age out of the replicated cache. While the read performance benefit of replication is, perhaps, smaller than expected, it is still significant in cases and will be mostly obtained in files that can remain replicated for a few days.

When encoding simplicity encourages the contents of each RAID set be taken from the same file, but because DISC file systems have such large block size, this leads to excessive capacity overhead. When RAID sets include blocks from multiple files, deletion of one file either does not reclaim space or requires check blocks to be recomputed. Data collected from DISC system at Yahoo! and Facebook guide our evaluation of these effects. Perhaps not surprisingly traditional RAID issues such as the small write performance penalty occur in DISC systems too.

Our prototype has been built for HDFS and released with the Apache project as MAPREDUCE-2036 for all users' benefit.

References

[BBBM95] Mario Blaum, Jim Brady, Jehoshua Bruck, and Jai Menon. EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures. *IEEE Transactions on Computers*, 44(2):192–202, 1995.

[BFK06] Ellie Berriman, Paul Feresten, and Shawn Kung. NetApp RAID-DP: Dual-Parity Raid 6 Protection Without Compromise. 2006.

[BFL$^+$01] Vasken Bohossian, Chenggong C. Fan, Paul S. LeMahieu, Marc D. Riedel, Lihao Xu, and Jehoshua Bruck. Computing in the RAIN: A Reliable Array of Independent Nodes. *IEEE Transactions on Parallel and Distributed Systems*, 12(2):99–114, 2001.

[BHS95] T. Blackwell, J. Harris, and M. Seltzer. Heuristic Cleaning Algorithms for Log-Structured File Systems. In *Proc. of the 1995 Winter USENIX Technical Conference*, pages 277–288, New Orleans, LA, January 1995.

[Bor09a] Dhruba Borthakur. HDFS and Erasure Codes, August 2009. http://hadoopblog.blogspot.com/2009/08/hdfs-and-erasure-codes-hdfs-raid.html.

[Bor09b] Dhruba Borthakur. The Hadoop Distributed File System: Architecture and Design, 2009. http://hadoop.apache.org/common/docs/current/hdfs_design.html.

[CEG$^+$04] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In *Proc. of the 2004 Conference on File and Storage Technologies*, pages 1–14, 2004.

[CG91] Vincent Cate and Thomas R. Gross. Integration of Compression and Caching for a Two-Level File System. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 200–211, April 1991.

[CK93] Thomas Cormen and David Kotz. Integrating Theory and Practice in Parallel File Systems. In *Proc of the 1993 DAGS/PC Symposium*, pages 64–74, 1993.

[CLG$^+$94] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, 1994.

[Col] Storage Architecture and Challenges. http://research.google.com/university/relations/facultysummit2010/storage_architecture_and_challenges.pdf.

[CWBLRT00] Philip H. Carns, III Walter B. Ligon, Robert B. Ross, and Rajeev Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proc. of the 4th Annual Linux Showcase and Conference*, pages 317–327, Berkeley, CA, USA, 2000.

[Day08] Shobhit Dayal. Characterizing HEC Storage Systems at Rest. Technical Report CMU-PDL-08-109, Carnegie Mellon University, 2008.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of the 6th Symposium on Operating System Design and Implementation*, pages 137–150, Berkeley, CA, USA, 2004.

[Fac] Facebook Datawarehouse Cluster. http://hadoopblog.blogspot.com/2010/05/facebook-has-worlds-largest-hadoop.html.

[FRL$^+$10] Bin Fu, Kai Ren, Julio Lopez, Eugene Fink, and Garth Gibson. DiscFinder: A Data-Intensive Scalable Cluster Finder for Astrophysics. In *Proc. of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 348–351, Chicago, IL, USA, June 2010.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *ACM SIGOPS Operating Systems Review*, 37(5):29–43, 2003.

[GHK+89] Garth A. Gibson, Lisa Hellerstein, Richard M. Karp, Randy H. Katz, and David A. Patterson. Failure Correction Techniques for Large Disk Arrays. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 123–132, 1989.

[Haf05] James Lee Hafner. WEAVER Codes: Highly Fault Tolerant Erasure Codes for Storage Systems. In *Proc. of the 2005 Conference on File and Storage Technologies*, 2005.

[HAR] Hadoop Archives Guide. http://hadoop.apache.org/common/docs/r0.20.0/hadoop_archives.html.

[HG92] Mark Holland and Garth A. Gibson. Parity Declustering for Continuous Operation in Redundant Disk Arrays. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 23–35, 1992.

[HKM+88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[HO93] J. Hartman and J. Ousterhout. The Zebra Striped Network File System. In *Proc. of the 14th ACM Symposium on Operating System Principles*, pages 29–43, 1993.

[KMF10] U Kang, Brendan Meeder, and Christos Faloutsos. Spectral Analysis for Billion-Scale Graphs: Discoveries and Implementation. In *Proc. of the 14th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, Hyderabad, India, June 2010.

[LMC94] Darrell D. E. Long, Bruce R. Montague, and Luis-Felipe Cabrera. Swift/RAID: A Distributed RAID System. *ACM Computing Systems*, (3):333–359, 1994.

[Lus] Lustre. http://www.lustre.org.

[MC93] Jai Menon and Jim Cortney. The Architecture of a Fault-Tolerant Cached RAID Controller. In *Proc of the 20th Annual International Symposium on Computer Architecture*, pages 76–86, 1993.

[ML90] Richard R. Muntz and John C.S. Lui. Performance Analysis of Disk Arrays Under Failure. In *Proc. of the 16th International Conference on Very Large Data Bases*, pages 162–173, 1990.

[NWO88] M. Nelson, B. Wlch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM Transaction on Computer Systems*, 6(1):134–154, February 1988.

[OCH+85] J. OUsterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proc. 10th ACM Symposium on Operating Systems Principles*, pages 15–24, 1985.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). *ACM SIGMOD Record*, 17(3):109–116, 1988.

[Pla97] James S. Plank. A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems. *Software - Practice & Experience*, 27(9):995–1012, September 1997.

[PLS+09] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries For Storage. In *Proc. of the 7th USENIX Conference on File and Storage Technologies*, pages 253–265, Februry 2009.

[PSS08] James S. Plank, Scott Simmerman, and Catherine D. Schuman. Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications - Version 1.2. Technical Report UT-CS-08-627, University of Tennessee Department of Computer Science, August 2008.

[RO91] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10:1–15, 1991.

[RS60] I. S. Reed and G. Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[S3] Amazon Simple Storage Service. `http://aws.amazon.com/s3`.

[SH02] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proc. of the 1st USENIX Conference on File and Storage Technologies*, page 19, Berkeley, CA, USA, 2002.

[SRO96] Steven R. Soltis, Thomas M. Ruwart, and Matthew T. O'Keefe. The Global File System. In *Proc. of the 5th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 319–342, 1996.

[SW96] Stefan Savage and John Wilkes. AFRAID: A Frequently Redundant Array of Independent Disks. In *Proc. of annual conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, 1996.

[Tan10] Wittawat Tantisiriroj. MAPREDUCE-2036: Enable Erasure Code in Tool Similar to Hadoop Archive, 2010. `https://issues.apache.org/jira/browse/MAPREDUCE-2036`.

[WGSS96] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, 1996.

[WK02] H. Weatherspoon and J. Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. In *Proc. of the 1st International workshop on Peer-To-Peer Systems*, March 2002.

[WUA$^+$08] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable Performance of the Panasas Parallel File System. In *Proc. of the 6th USENIX Conference on File and Storage Technologies*, pages 17–33, 2008.

[ZDM$^+$10] Zhe Zhang, Amey Deshpande, Xiaosong Ma, Eno Thereska, , and Dushyanth Narayanan. Does Erasure Coding Have a Role to Play in My Data Center? Technical Report MSR-TR-2010-52, Microsoft Research, 2010.