USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Analysis of HDFS Under HBase: A Facebook Messages Case Study

Tyler Harter, *University of Wisconsin—Madison;* Dhruba Borthakur, Siying Dong, Amitanand Aiyer, and Liyin Tang, *Facebook Inc.;* Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau, *University of Wisconsin—Madison*

https://www.usenix.org/conference/fast14/technical-sessions/presentation/harter

This paper is included in the Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST '14).

February 17–20, 2014 • Santa Clara, CA USA

ISBN 978-1-931971-08-9

Open access to the Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST '14) is sponsored by

Symantec

Analysis of HDFS Under HBase: A Facebook Messages Case Study

Tyler Harter, Dhruba Borthakur†, Siying Dong†, Amitanand Aiyer†,
Liyin Tang†, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

University of Wisconsin, Madison † Facebook Inc.

Abstract

We present a multilayer study of the Facebook Messages stack, which is based on HBase and HDFS. We collect and analyze HDFS traces to identify potential improvements, which we then evaluate via simulation. Messages represents a new HDFS workload: whereas HDFS was built to store very large files and receive mostly-sequential I/O, 90% of files are smaller than 15MB and I/O is highly random. We find hot data is too large to easily fit in RAM and cold data is too large to easily fit in flash; however, cost simulations show that adding a small flash tier improves performance more than equivalent spending on RAM or disks. HBase's layered design offers simplicity, but at the cost of performance; our simulations show that network I/O can be halved if compaction bypasses the replication layer. Finally, although Messages is read-dominated, several features of the stack (i.e., logging, compaction, replication, and caching) amplify write I/O, causing writes to dominate disk I/O.

1 Introduction

Large-scale distributed storage systems are exceedingly complex and time consuming to design, implement, and operate. As a result, rather than cutting new systems from whole cloth, engineers often opt for *layered* architectures, building new systems upon already-existing ones to ease the burden of development and deployment.

Layering, as is well known, has many advantages [23]. For example, construction of the Frangipani distributed file system [27] was greatly simplified by implementing it atop Petal [19], a distributed and replicated block-level storage system. Because Petal provides scalable, fault-tolerant virtual disks, Frangipani could focus solely on file-system level issues (*e.g.*, locking); the result of this two-layer structure, according to the authors, was that Frangipani was "relatively easy to build" [27].

Unfortunately, layering can also lead to problems, usually in the form of decreased performance, lowered reliability, or other related issues. For example, Denehy *et al.* show how naïve layering of journaling file systems atop software RAIDs can lead to data loss or corruption [5]. Similarly, others have argued about the general inefficiency of the file system atop block devices [10].

In this paper, we focus on one specific, and increasingly common, layered storage architecture: a distributed database (HBase, derived from BigTable [3]) atop a distributed file system (HDFS [24], derived from the Google File System [11]). Our goal is to study the interaction of these important systems, with a particular focus on the lower layer; thus, our highest-level question: is HDFS an effective storage backend for HBase?

To derive insight into this hierarchical system, and thus answer this question, we trace and analyze it under a popular workload: Facebook Messages (FM) [20]. FM is a messaging system that enables Facebook users to send chat and email-like messages to one another; it is quite popular, handling millions of messages each day. FM stores its information within HBase (and thus, HDFS), and hence serves as an excellent case study.

To perform our analysis, we first collect detailed HDFS-level traces over an eight-day period on a subset of machines within a specially-configured *shadow cluster*. FM traffic is mirrored to this shadow cluster for the purpose of testing system changes; here, we utilize the shadow to collect detailed HDFS traces. We then analyze said traces, comparing results to previous studies of HDFS under more traditional workloads [14, 16].

To complement to our analysis, we also perform numerous simulations of various caching, logging, and other architectural enhancements and modifications. Through simulation, we can explore a range of "what if?" scenarios, and thus gain deeper insight into the efficacy of the layered storage system.

Overall, we derive numerous insights, some expected and some surprising, from our combined analysis and simulation study. From our analysis, we find writes represent 21% of I/O to HDFS files; however, further investigation reveals the vast majority of writes are HBase overheads from logging and compaction. Aside from these overheads, FM writes are scarce, representing only 1% of the "true" HDFS I/O. Diving deeper in the stack, simulations show writes become amplified. Beneath HDFS replication (which triples writes) and OS caching (which absorbs reads), 64% of the final disk load is write I/O. This write blowup (from 1% to 64%) emphasizes the importance of optimizing writes in layered systems, even for especially read-heavy workloads like FM.

From our simulations, we further extract the following conclusions. We find that caching at the DataNodes is still (surprisingly) of great utility; even at the last layer of the storage stack, a reasonable amount of memory per node (*e.g.*, 30GB) significantly reduces read load. We also find that a "no-write allocate" policy generally performs best, and that higher-level hints regarding writes only provide modest gains. Further analysis shows the utility of server-side flash caches (in addition to RAM), *e.g.*, adding a 60GB SSD can reduce latency by 3.5x.

Finally, we evaluate the effectiveness of more substantial HDFS architectural changes, aimed at improving write handling: local compaction and combined logging. Local compaction performs compaction work within each replicated server instead of reading and writing data across the network; the result is a 2.7x reduction in network I/O. Combined logging consolidates logs from multiple HBase RegionServers into a single stream, thus reducing log-write latencies by 6x.

The rest of this paper is organized as follows. First, a background section describes HBase and the Messages storage architecture (§2). Then we describe our methodology for tracing, analysis, and simulation (§3). We present our analysis results (§4), make a case for adding a flash tier (§5), and measure layering costs (§6). Finally, we discuss related work (§7) and conclude (§8).

2 Background

We now describe the HBase sparse-table abstraction (§2.1) and the overall FM storage architecture (§2.2).

2.1 Versioned Sparse Tables

HBase, like BigTable [3], provides a *versioned sparse-table* interface, which is much like an associative array, but with two major differences: (1) keys are ordered, so lexicographically adjacent keys will be stored in the same area of physical storage, and (2) keys have semantic meaning which influences how HBase treats the data. Keys are of the form *row:column:version*. A *row* may be any byte string, while a *column* is of the form *family:name*. While both column families and names may be arbitrary strings, families are typically defined statically by a schema while new column names are often created during runtime. Together, a row and column specify a cell, for which there may be many versions.

A sparse table is sharded along both row and column dimensions. Rows are grouped into *regions*, which are responsible for all the rows within a given row-key range. Data is sharded across different machines with region granularity. Regions may be split and re-assigned to machines with a utility or automatically upon reboots. Columns are grouped into families so that the application may specify different policies for each group (*e.g.*, what compression to use). Families also provide a locality hint: HBase clusters together data of the same family.

2.2 Messages Architecture

Users of FM interact with a web layer, which is backed by an application cluster, which in turn stores data in a separate HBase cluster. The application cluster executes FM-specific logic and caches HBase rows while HBase itself is responsible for persisting most data. Large objects (*e.g.*, message attachments) are an exception; these are stored in Haystack [25] because HBase is inefficient for large data (§4.1). This design applies Lampson's advice to "handle normal and worst case separately" [18].

HBase stores its data in HDFS [24], a distributed file system which resembles GFS [11]. HDFS triply replicates data in order to provide availability and tolerate failures. These properties free HBase to focus on higher-level database logic. Because HBase stores all its data in HDFS, the same machines are typically used to run both HBase and HDFS servers, thus improving locality. These clusters have three main types of machines: an *HBase master*, an *HDFS NameNode*, and many *worker* machines. Each worker runs two servers: an *HBase RegionServer* and an *HDFS DataNode*. HBase clients use the HBase master to map row keys to the *one* RegionServer responsible for that key. Similarly, an HDFS NameNode helps HDFS clients map a pathname and block number to the *three* DataNodes with replicas of that block.

3 Methodology

We now discuss trace collection and analysis (§3.1), simulation (§3.2), validity (§3.3), and confidentiality (§3.4).

3.1 Trace Collection and Analysis

Prior Hadoop trace studies [4, 16] typically analyze default MapReduce or HDFS logs, which record coarse-grained file events (*e.g.*, creates and opens), but lack details about individual requests (*e.g.*, offsets and sizes). For our study, we build a new trace framework, HTFS (Hadoop Trace File System) to collect these details. Some data, though (*e.g.*, the contents of a write), is not recorded; this makes traces smaller and (more importantly) protects user privacy.

HTFS extends the HDFS client library, which supports the arbitrary composition of layers to obtain a desired feature set (*e.g.*, a checksumming layer may be used). FM deployments typically have two layers: one for normal NameNode and DataNode interactions, and one for fast failover [6]. HDFS clients (*e.g.*, RegionServers) can record I/O by composing HTFS with other layers. HTFS can trace over 40 HDFS calls and is publicly available with the Facebook branch of Hadoop.[1]

[1] `https://github.com/facebook/hadoop-20/blob/master/src/hdfs/org/apache/hadoop/hdfs/APITraceFileSystem.java`

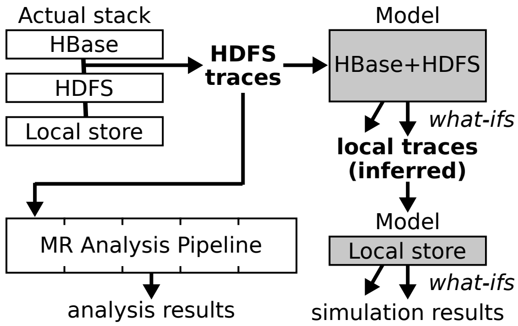

Figure 1: **Tracing, analysis, and simulation.**

We collect our traces on a specially configured *shadow cluster* that receives the same requests as a production FM cluster. Facebook often uses shadow clusters to test new code before broad deployment. By tracing in an HBase/HDFS shadow cluster, we were able to study the real workload without imposing overheads on real users. For our study, we randomly selected nine worker machines, configuring each to use HTFS.

We collected traces for 8.3 days, starting June 7, 2013. We collected 116GB of gzip-compressed traces, representing 5.2 billion recorded events and 71TB of HDFS I/O. The machines each had 32 Xeon(R) CPU cores and 48GB of RAM, 16.4GB of which was allocated for the HBase cache (most memory is left to the file-system cache, as attempts to use larger caches in HBase cause JVM garbage-collection stalls). The HDFS workload is the product of a 60/34/6 get/put/delete ratio for HBase.

As Figure 1 shows, the traces enable both analysis and simulation. We analyzed our traces with a pipeline of 10 MapReduce jobs, each of which transforms the traces, builds an index, shards events, or outputs statistics. Complex dependencies between events require careful sharding for correctness. For instance, a stream-open event and a stream-write event must be in the same compute shard in order to correlate I/O with file type. Furthermore, sharding must address the fact that different paths may refer to the same data (due to renames).

3.2 Modeling and Simulation

We evaluate changes to the storage stack via simulation. Our simulations are based on two models (illustrated in Figure 1): a model which determines how the HDFS I/O translates to local I/O and a model of local storage.

How HDFS I/O translates to local I/O depends on several factors, such as prior state, replication policy, and configurations. Making all these factors match the actual deployment would be difficult, and modeling what happens to be the current configuration is not particularly interesting. Thus, we opt for a model which is easy to understand and plausible (*i.e.*, it reflects a hypothetical policy and state which could reasonably occur).

Our model assumes the HDFS files in our traces are replicated by nine DataNodes which co-reside with the nine RegionServers we traced. The data for each RegionServer is replicated to one co-resident and two remote DataNodes. HDFS file blocks are 256MB in size; thus, when a RegionServer writes a 1GB HDFS file, our model translates that to the creation of twelve 256MB local files (four per replica). Furthermore, 2GB of network reads are counted for the remote replicas. This simplified model of replication could lead to errors for load balancing studies, but we believe little generality is lost for caching simulations and our other experiments. In production, all the replicas of a RegionServer's data may be remote (due to region re-assignment), causing additional network I/O; however, long-running FM-HBase clusters tend to converge over time to the pattern we simulate.

The HDFS+HBase model's output is the input for our local-store simulator. Each local store is assumed to have an HDFS DataNode, a set of disks (each with its own file system and disk scheduler), a RAM cache, and possibly an SSD. When the simulator processes a request, a balancer module representing the DataNode logic directs the request to the appropriate disk. The file system for that disk checks the RAM and flash caches; upon a miss, the request is passed to a disk scheduler for re-ordering.

The scheduler switches between files using a round-robin policy (1MB slice). The C-SCAN policy [1] is then used to choose between multiple requests to the same file. The scheduler dispatches requests to a disk module which determines latency. Requests to different files are assumed to be distant, and so require a 10ms seek. Requests to adjacent offsets of the same file, however, are assumed to be adjacent on disk, so blocks are transferred at 100MB/s. Finally, we assume some locality between requests to non-adjacent offsets in the same file; for these, the seek time is $min\{10ms, distance/(100MB/s)\}$.

3.3 Simulation Validity

We now address three validity questions: *does ignoring network latency skew our results? Did we run our simulations long enough? Are simulation results from a single representative machine meaningful?*

First, we explore our assumption about constant network latency by adding random jitter to the timing of requests and observing how important statistics change. Table 1 shows how much error results by changing request issue times by a uniform-random amount. Errors are very small for 1ms jitter (at most 1.3% error). Even with a 10ms jitter, the worst error is 6.6%. Second, in order to verify that we ran the simulations long enough, we measure how the statistics would have been different if we had finished our simulations 2 or 4 days earlier (in-

statistic	baseline	jitter ms 1	5	10	finish day -2	-4	sample median
FS reads MB/min	576	0.0	0.0	0.0	-3.4	-0.6	-4.2
FS writes MB/min	447	0.0	0.0	0.0	-7.7	-11.5	-0.1
RAM reads MB/min	287	-0.0	0.0	0.0	-2.6	-2.4	-6.2
RAM writes MB/min	345	0.0	-0.0	-0.0	-3.9	1.1	-2.4
Disk reads MB/min	345	-0.0	0.0	0.0	-3.9	1.1	-2.4
Disk writes MB/min	616	-0.0	1.3	1.9	-5.3	-8.3	-0.1
Net reads MB/min	305	0.0	0.0	0.0	-8.7	-18.4	-2.8
Disk reqs/min	275.1K	0.0	0.0	0.0	-4.6	-4.7	-0.1
(user-read)	65.8K	0.0	-0.0	-0.0	-2.9	-0.8	-4.3
(log)	104.1K	0.0	0.0	0.0	1.6	1.3	-1.0
(flush)	4.5K	0.0	0.0	0.0	1.2	0.4	-1.3
(compact)	100.6K	-0.0	-0.0	-0.0	-12.2	-13.6	-0.1
Disk queue ms	6.17	-0.4	-0.5	-0.0	-3.2	0.6	-1.8
(user-read)	12.3	0.1	-0.8	-1.8	-0.2	2.7	1.7
(log)	2.47	-1.3	-1.1	0.6	-4.9	-6.4	-6.0
(flush)	5.33	0.3	0.0	-0.3	-2.8	-2.6	-1.0
(compact)	6.0	-0.6	0.0	2.0	-3.5	2.5	-6.4
Disk exec ms	0.39	0.1	1.0	2.5	1.0	2.0	-1.4
(user-read)	0.84	-0.1	-0.5	-0.7	-0.0	-0.1	-1.2
(log)	0.26	0.4	3.3	6.6	-2.1	-1.7	0.0
(flush)	0.15	-0.3	0.7	3.2	-1.1	-0.9	-0.8
(compact)	0.24	-0.0	2.1	5.2	4.0	4.8	-0.3

Table 1: **Statistic Sensitivity.** *The first column group shows important statistics and their values for a representative machine. Other columns show how these values would change (as percentages) if measurements were done differently. Low percentages indicate a statistic is robust.*

stead of using the full 8.3 days of traces). The differences are worse than for jitter, but are still usually small, and are at worst 18.4% for network I/O.

Finally, we evaluate whether it is reasonable to pick a single representative instead of running our experiments for all nine machines in our sample. Running all our experiments for a single machine alone takes about 3 days on a 24-core machine with 72GB of RAM, so basing our results on a representative is desirable. The final column of Table 1 compares the difference between statistics for our representative machine and the median of statistics for all nine machines. Differences are quite small and are never greater than 6.4%, so we use the representative for the remainder of our simulations (trace-analysis results, however, will be based on all nine machines).

3.4 Confidentiality

In order to protect user privacy, our traces only contain the sizes of data (*e.g.*, request and file sizes), but never actual data contents. Our tracing code was carefully reviewed by Facebook employees to ensure compliance with Facebook privacy commitments. We also avoid presenting commercially-sensitive statistics, such as would allow estimation of the number of users of the service. While we do an in-depth analysis of the I/O patterns on a sample of machines, we do not disclose how large the sample is as a fraction of all the FM clusters. Much of the architecture we describe is open source.

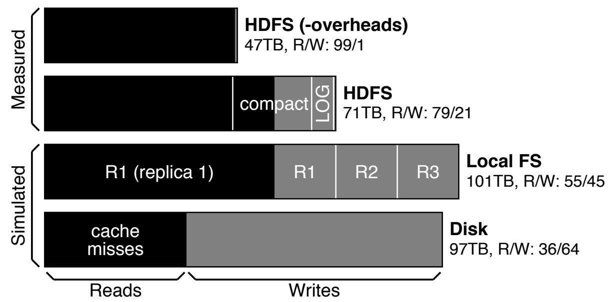

Figure 2: **I/O across layers.** *Black sections represent reads and gray sections represent writes. The top two bars indicate HDFS I/O as measured directly in the traces. The bottom two bars indicate local I/O at the file-system and disk layers as inferred via simulation.*

4 Workload Behavior

We now characterize the FM workload with four questions: *what are the major causes of I/O at each layer of the stack (§4.1)? How much I/O and space is required by different types of data (§4.2)? How large are files, and does file size predict file lifetime (§4.3)?* And *do requests exhibit patterns such as locality or sequentiality (§4.4)?*

4.1 Multilayer Overview

We begin by considering the number of reads and writes at each layer of the stack in Figure 2. At a high level, FM issues `put()` and `get()` requests to HBase. The `put` data accumulates in buffers, which are occasionally flushed to *HFiles* (HDFS files containing sorted key-value pairs and indexing metadata). Thus, `get` requests consult the write buffers as well as the appropriate HFiles in order to retrieve the most up-to-date value for a given key. This core I/O (`put`-flushes and `get`-reads) is shown in the first bar of Figure 2; the 47TB of I/O is 99% reads.

In addition to the core I/O, HBase also does logging (for durability) and compaction (to maintain a read-efficient layout) as shown in the second bar. Writes account for most of these overheads, so the R/W (read/write) ratio decreases to 79/21. Flush data is compressed but log data is not, so logging causes 10x more writes even though the same data is both logged and flushed. Preliminary experiments with log compression [26] have reduced this ratio to 4x. Flushes, which can be compressed in large chunks, have an advantage over logs, which must be written as `puts` arrive. Compaction causes about 17x more writes than flushing does, indicating that a typical piece of data is relocated 17 times. FM stores very large objects (*e.g.*, image attachments) in Haystack [17] for this reason. FM is a very read-heavy HBase workload within Facebook, so it is tuned to compact aggressively. Compaction makes reads faster by merge-sorting many small HFiles into fewer big HFiles,

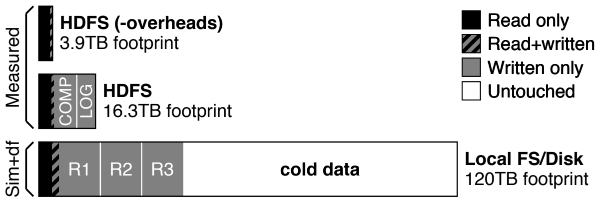

Figure 3: **Data across layers.** *This is the same as Figure 2 but for data instead of I/O. COMP is compaction.*

Family	Description
Actions	Log of user actions and message contents
MessageMeta	Metadata per message (*e.g.*, isRead and subject)
ThreadMeta	Metadata per thread (*e.g.*list of participants)
PrefetchMeta	Privacy settings, contacts, mailbox summary, etc.
Keywords	Word-to-message map for search and typeahead
ThreaderThread	Thread-to-message mapping
ThreadingIdIdx	Map between different types of message IDs
ActionLogIdIdx	Also a message-ID map (like ThreadingIdIdx)

Table 2: **Schema.** *HBase column families are described.*

thus reducing the number of files a `get` must check.

FM tolerates failures by replicating data with HDFS. Thus, writing an HDFS block involves writing three local files and two network transfers. The third bar of Figure 2 shows how this tripling further reduces the R/W ratio to 55/45. Furthermore, OS caching prevents some of these file-system reads from hitting disk. With a 30GB cache, the 56TB of reads at the file-system level cause only 35TB of reads at the disk level, as shown in the fourth bar. Also, very small file-system writes cause 4KB-block disk writes, so writes are increased at the disk level. Because of these factors, writes represent 64% of disk I/O.

Figure 3 gives a similar layered overview, but for data rather than I/O. The first bar shows 3.9TB of HDFS data received some core I/O during tracing (data deleted during tracing is not counted). Nearly all this data was read and a small portion written. The second bar also includes data which was accessed only by non-core I/O; non-core data is several times bigger than core data. The third bar shows how much data is touched at the local level during tracing. This bar also shows *untouched* data; we estimate[2] this by subtracting the amount of data we infer was touched due to HDFS I/O from the disk utilization (measured with `df`). Most of the 120TB of data is very cold; only a third is accessed over the 8-day period.

Conclusion: FM is very read-heavy, but logging, compaction, replication, and caching amplify write I/O, causing writes to dominate disk I/O. We also observe that while the HDFS dataset accessed by core I/O is relatively small, on disk the dataset is very large (120TB) and very cold (two thirds is never touched). Thus, architectures to support this workload should consider its hot/cold nature.

4.2 Data Types

We now study the types of data FM stores. Each user's data is stored in a single HBase row; this prevents the data from being split across different RegionServers. New data for a user is added in new columns within the row. Related columns are grouped into families, which are defined by the FM schema (summarized in Table 2).

The *Actions* family is a log built on top of HBase, with different log records stored in different columns; *addMsg* records contain actual message data while other records (*e.g.*, *markAsRead*) record changes to metadata state. Getting the latest state requires reading a number of recent records in the log. To cap this number, a metadata snapshot (a few hundred bytes) is sometimes written to the *MessageMeta* family. Because Facebook chat is built over messages, metadata objects are large relative to many messages (*e.g.*, "hey, whasup?"). Thus, writing a change to *Actions* is generally much cheaper than writing a full metadata object to *MessageMeta*. Other metadata is stored in *ThreadMeta* and *PrefetchMeta* while *Keywords* is a keyword-search index and *ThreaderThread*, *ThreadingIdIdx*, and *ActionLogIdIdx* are other indexes.

Figure 4a shows how much data of each type is accessed at least once during tracing (including later-deleted data); a total (sum of bars) of 26.5TB is accessed. While actual messages (*i.e.*, *Actions*) take significant space, helper data (*e.g.*, metadata, indexes, and logs) takes much more. We also see that little data is both read and written, suggesting that writes should be cached selectively (if at all). Figure 4b reports the I/O done for each type. We observe that some families receive much more I/O per data, *e.g.*, an average data byte of *PrefetchMeta* receives 15 bytes of I/O whereas a byte of *Keywords* receives only 1.1.

Conclusion: FM uses significant space to store messages and does a significant amount of I/O on these messages; however, both space and I/O are dominated by helper data (*i.e.*, metadata, indexes, and logs). Relatively little data is both written and read during tracing; this suggests caching writes is of little value.

4.3 File Size

GFS (the inspiration for HDFS) assumed that "multi-GB files are the common case, and should be handled efficiently" [11]. Other workload studies confirm this, *e.g.*, MapReduce inputs were found to be about 23GB at the 90th percentile (Facebook in 2010) [4]. We now revisit the assumption that HDFS files are large.

Figure 5 shows, for each file type, a distribution of file sizes (about 862 thousand files appear in our traces). Most files are small; for each family, 90% are smaller

[2]the RegionServers in our sample store some data on DataNodes outside our sample (and vice versa), so this is a sample-based estimate rather than a direct correlation of HDFS data to disk data

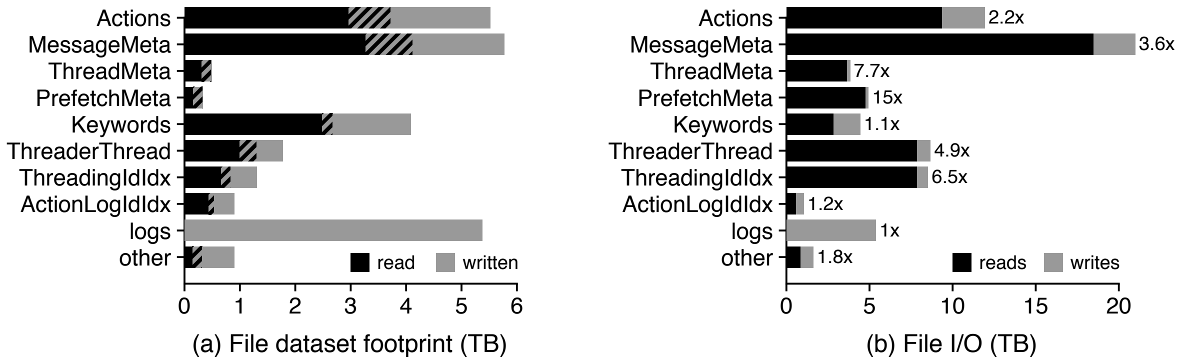

Figure 4: **File types.** *Left: all accessed HDFS file data is broken down by type. Bars further show whether data was read, written, or both. Right: I/O is broken down by file type and read/write. Bar labels indicate the I/O-to-data ratio.*

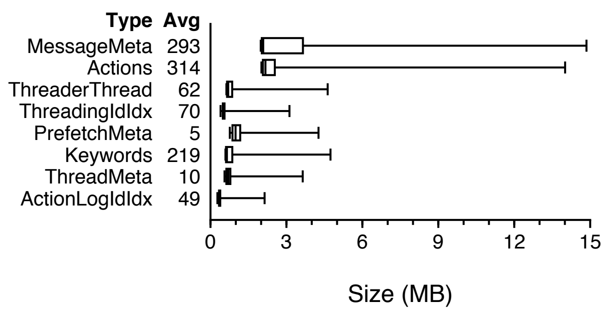

Figure 5: **File-size distribution.** *This shows a box-and-whiskers plot of file sizes. The whiskers indicate the 10th and 90th percentiles. On the left, the type of file and average size is indicated. Log files are not shown, but have an average size of 218MB with extremely little variance.*

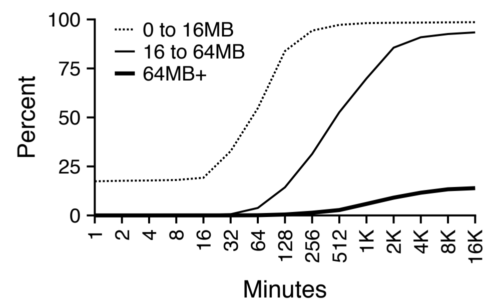

Figure 6: **Size/life correlation.** *Each line is a CDF of lifetime for created files of a particular size. Not all lines reach 100% as some files are not deleted during tracing.*

than 15MB. However, a handful are so large as to skew averages upwards significantly, *e.g.*, the average MessageMeta file is 293MB.

Although most files are very small, compaction should quickly replace these small files with a few large, long-lived files. We divide files created during tracing into small (0 to 16MB), medium (16 to 64MB), and large (64MB+) categories. 94% of files are small, 2% are medium, and 4% are large; however, large files contain 89% of the data. Figure 6 shows the distribution of file lifetimes for each category. 17% of small files are deleted within less than a minute, and very few last more than a few hours; about half of medium files, however, last more than 8 hours. Only 14% of the large files created during tracing were also deleted during tracing.

Conclusion: Traditional HDFS workloads operate on very large files. While most FM data lives in large, long-lived files, most files are small and short-lived. This has metadata-management implications; HDFS manages all file metadata with a single NameNode because the data-to-metadata ratio is assumed to be high. For FM, this assumption does not hold; perhaps distributing HDFS metadata management should be reconsidered.

4.4 I/O Patterns

We explore three relationships between different read requests: temporal locality, spatial locality, and sequentiality. We use a new type of plot, a *locality map*, that describes all three relationships at once. Figure 7 shows a locality map for FM reads. The data shows how often a read was *recently* preceded by a *nearby* read, for various thresholds on "recent" and "nearby". Each line is a hit-ratio curve, with the x-axis indicating how long items are cached. Different lines represent different levels of prefetching, *e.g.*, the 0-line represents no prefetching, whereas the 1MB-line means data 1MB before and 1MB after a read is prefetched.

Line shape describes *temporal locality*, *e.g.*, the 0-line gives a distribution of time intervals between different reads to the same data. Reads are almost never preceded by a prior read to the same data in the past four minutes; however, 26% of reads are preceded within the last 32 minutes. Thus, there is significant temporal locality (*i.e.*, reads are near each other with respect to time), and additional caching should be beneficial. The locality map also shows there is little *sequentiality*. A highly sequen-

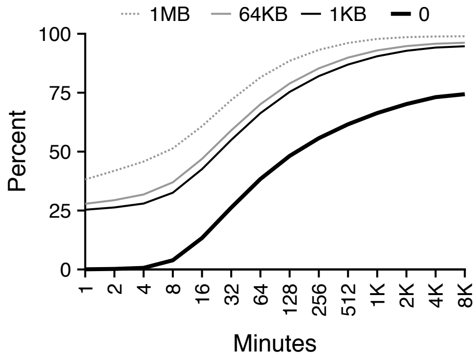

Figure 7: **Reads: locality map.** *This plot shows how often a read was recently preceded by a nearby read, with time-distance represented along the x-axis and offset-distance represented by the four lines.*

tial pattern would show that many reads were recently preceded by I/O to nearby offsets; here, however, the 1KB-line shows only 25% of reads were preceded by I/O to very nearby offsets within the last minute. Thus, over 75% of reads are random. The distances between the lines of the locality map describe *spatial locality*. The 1KB-line and 64KB-line are very near each other, indicating that (except for sequential I/O) reads are rarely preceded by other reads to nearby offsets. This indicates very low spatial locality (*i.e.*, reads are far from each other with respect to offset), and additional prefetching is unlikely to be helpful.

To summarize the locality map, the main pattern reads exhibit is temporal locality (there is little sequentiality or spatial locality). High temporal locality implies a significant portion of reads are "repeats" to the same data. We explore this repeated-access pattern further in Figure 8a. The bytes of HDFS file data that are read during tracing are distributed along the x-axis by the number of reads. The figure shows that most data (73.7%) is read only once, but 1.1% of the data is read at least 64 times. Thus, repeated reads are not spread evenly, but are concentrated on a small subset of the data.

Figure 8b shows how many bytes are read for each of the categories of Figure 8a. While 19% of the reads are to bytes which are only read once, most I/O is to data which is accessed many times. Such bias at this level is surprising considering that all HDFS I/O has missed two higher-level caches (an application cache and the HBase cache). Caches are known to lessen I/O to particularly hot data, *e.g.*, a multilayer photo-caching study found caches cause "distributions [to] flatten in a significant way" [15]. The fact that bias remains despite caching suggests the working set may be too large to fit in a small cache; a later section (§5.1) shows this to be the case.

Conclusion: At the HDFS level, FM exhibits relatively little sequentiality, suggesting high-bandwidth,

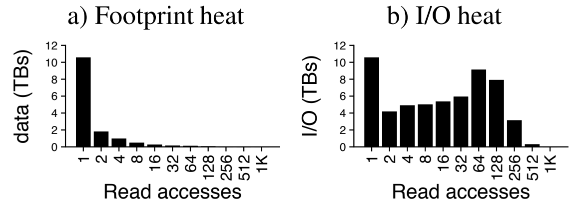

Figure 8: **Read heat.** *In both plots, bars show a distribution across different levels of read heat (i.e., the number of times a byte is read). The left shows a distribution of the dataset (so the bars sum to the dataset size, included deleted data), and the right shows a distribution of I/O to different parts of the dataset (so the bars sum to the total read I/O).*

high-latency storage mediums (*e.g.*, disk) are not ideal for serving reads. The workload also shows very little spatial locality, suggesting additional prefetching would not help, possibly because FM already chooses for itself what data to prefetch. However, despite application-level and HBase-level caching, some of the HDFS data is particularly hot; thus, additional caching could help.

5 Tiered Storage: Adding Flash

We now make a case for adding a flash tier to local machines. FM has a very large, mostly cold dataset (§4.1); keeping all this data in flash would be wasteful, costing upwards of $10K/machine[3]. We evaluate the two alternatives: use some flash or no flash. We consider four questions: *how much can we improve performance without flash, by spending more on RAM or disks (§5.1)? What policies utilize a tiered RAM/flash cache best (§5.2)? Is flash better used as a cache to absorb reads or as a buffer to absorb writes (§5.3)?* And ultimately, *is the cost of a flash tier justifiable (§5.4)?*

5.1 Performance without Flash

Can buying faster disks or more disks significantly improve FM performance? Figure 9 presents average disk latency as a function of various disk factors. The first plot shows that for more than 15 disks, adding more disks has quickly diminishing returns. The second shows that higher-bandwidth disks also have relatively little advantage, as anticipated by the highly-random workload observed earlier (§4.4). However, the third plot shows that latency is a major performance factor.

The fact that lower latency helps more than having additional disks suggests the workload has relatively little parallelism, *i.e.*, being able to do a few things quickly is better than being able to do many things at once. Un-

[3] at $0.80/GB, storing 13.3TB (120TB split over 9 machines) in flash would cost $10,895/machine.

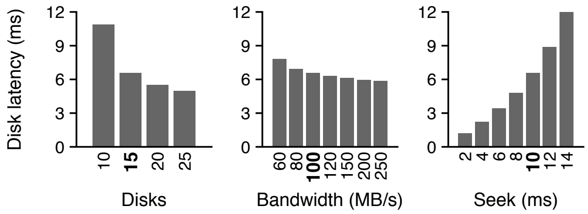

Figure 9: **Disk performance.** *The figure shows the relationship between disk characteristics and the average latency of disk requests. As a default, we use 15 disks with 100MB/s bandwidth and 10ms seek time. Each of the plots varies one of the characteristics, keeping the other two fixed.*

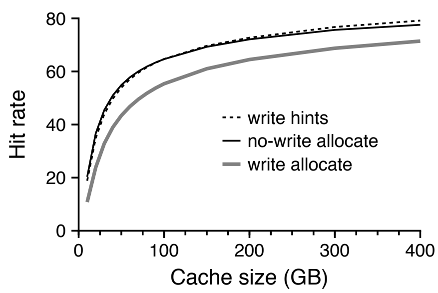

Figure 10: **Cache hit rate.** *The relationship between cache size and hit rate is shown for three policies.*

fortunately, the 2-6ms disks we simulate are unrealistically fast, having no commercial equivalent. Thus, although significant disk capacity is needed to store the large, mostly cold data, reads are better served by a low-latency medium (*e.g.*, RAM or flash).

Thus, we ask, *can the hot data fit comfortably in a pure-RAM cache?* We measure hit rate for cache sizes in the 10-400GB range. We also try three different LRU policies: *write allocate*, *no-write allocate*, and *write hints*. All three are write-through caches, but differ regarding whether written data is cached. Write allocate adds all write data, no-write allocate adds no write data, and the hint-based policy takes suggestions from HBase and HDFS. In particular, a written file is only cached if (a) the local file is a primary replica of the HDFS block, and (b) the file is either flush output (as opposed to compaction output) or is likely to be compacted soon.

Figure 10 shows, for each policy, that the hit rate increases significantly as the cache size increases up until about 200GB, where it starts to level off (but not flatten); this indicates the working set is very large. Earlier (§4.2), we found little overlap between writes and reads and concluded that written data should be cached selectively if at all. Figure 10 confirms: caching all writes is the worst policy. Up until about 100GB, "no-write allocate" and "write hints" perform about equally well. Beyond 100GB, hints help, but only slightly. We use no-write allocate throughout the remainder of the paper because it is simple and provides decent performance.

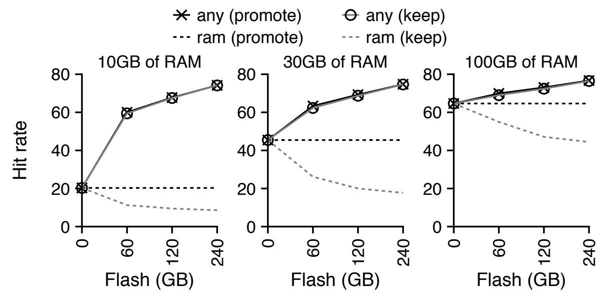

Figure 11: **Tiered hit rates.** *Overall hit rate (any) is shown by the solid lines for the promote and keep policies. The results are shown for varying amounts of RAM (different plots) and varying amounts of flash (x-axis). RAM hit rates are indicated by the dashed lines.*

Conclusion: The FM workload exhibits relatively little sequentiality or parallelism, so adding more disks or higher-bandwidth disks is of limited utility. Fortunately, the same data is often repeatedly read (§4.4), so a very large cache (*i.e.*, a few hundred GBs in size) can service nearly 80% of the reads. The usefulness of a very large cache suggests that storing at least some of the hot data in flash may be most cost effective. We evaluate the cost/performance tradeoff between pure-RAM and hybrid caches in a later section (§5.4).

5.2 Flash as Cache

In this section, we use flash as a second caching tier beneath RAM. Both tiers independently are LRU. Initial inserts are to RAM, and RAM evictions are inserted into flash. We evaluate exclusive cache policies. Thus, upon a flash hit, we have two options: the *promote policy* (PP) repromotes the item to the RAM cache, but the *keep policy* (KP) keeps the item at the flash level. PP gives the combined cache LRU behavior. The idea behind KP is to limit SSD wear by avoiding repeated promotions and evictions of items between RAM and flash.

Figure 11 shows the hit rates for twelve flash/RAM mixes. For example, the middle plot shows what the hit rate is when there is 30GB of RAM: without any flash, 45% of reads hit the cache, but with 60GB of flash, about 63% of reads hit in either RAM or flash (regardless of policy). The plots show that across all amounts of RAM and flash, the number of reads that hit in "any" cache differs very little between policies. However, PP causes significantly more of these hits to go to RAM; thus, PP will be faster because RAM hits are faster than flash hits.

We now test our hypothesis that, in trade for decreasing RAM hits, KP improves flash lifetime. We compute lifetime by measuring flash writes, assuming the FTL provides even wear leveling, and assuming the SSD supports 10K program/erase cycles. Figure 12 reports flash lifetime as the amount of flash varies along the x-axis.

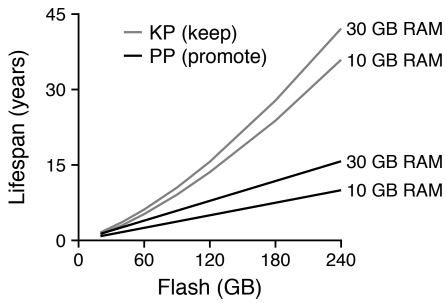

Figure 12: **Flash lifetime.** *The relationship between flash size and flash lifetime is shown for both the keep policy (gray lines) and promote policy (black lines). There are two lines for each policy (10 or 30GB RAM).*

The figure shows that having more RAM slightly improves flash lifetime. This is because flash writes occur upon RAM evictions, and evictions will be less frequent with ample RAM. Also, as expected, KP often doubles or triples flash lifetime, *e.g.*, with 10GB of RAM and 60GB of flash, using KP instead of PP increases lifetime from 2.5 to 5.2 years. The figure also shows that flash lifetime increases with the amount of flash. For PP, the relationship is perfectly linear. The number of flash writes equals the number of RAM evictions, which is independent of flash size; thus, if there is twice as much flash, each block of flash will receive exactly half as much wear. For KP, however, the flash lifetime increases superlinearly with size; with 10GB of RAM and 20GB of flash, the years-to-GB ratio is 0.06, but with 240GB of flash, the ratio is 0.15. The relationship is superlinear because additional flash absorbs more reads, causing fewer RAM inserts, causing fewer RAM evictions, and ultimately causing fewer flash writes. Thus, doubling the flash size decreases total flash writes in addition to spreading the writes over twice as many blocks.

Flash caches have an additional advantage: crashes do not cause cache contents to be lost. We quantify this benefit by simulating four crashes at different times and measuring changes to hit rate. Figure 13 shows the results of two of these crashes for 100GB caches with different flash-to-RAM ratios (using PP). Even though the hottest data will be in RAM, keeping some data in flash significantly improves the hit rate after a crash. The examples also show that it can take 4-6 hours to fully recover from a crash. We quantify the total recovery cost in terms of additional disk reads (not shown). Whereas crashing with a pure-RAM cache on average causes 26GB of additional disk I/O, crashing costs only 10GB for a hybrid cache which is 75% flash.

Conclusion: Adding flash to RAM can greatly improve the caching hit rate; furthermore (due to persistence) a hybrid flash/RAM cache can eliminate half of the extra disk reads that usually occur after a crash. However, using flash raises concerns about wear. Shuffling data between flash and RAM to keep the hottest data in RAM improves performance but can easily decrease SSD lifetime by a factor of 2x relative to a wear-aware policy. Fortunately, larger SSDs tend to have long lifetimes for FM, so wear may be a small concern (*e.g.*, 120GB+ SSDs last over 5 years regardless of policy).

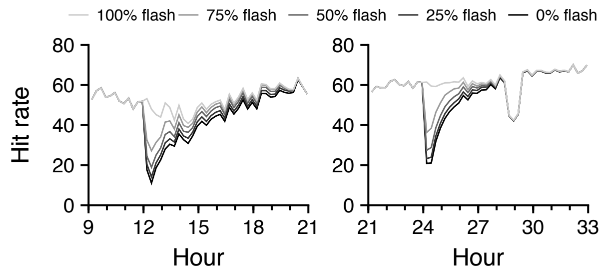

Figure 13: **Crash simulations.** *The plots show two examples of how crashing at different times affects different 100GB tiered caches, some of which are pure flash, pure RAM, or a mix. Hit rates are unaffected when crashing with 100% flash.*

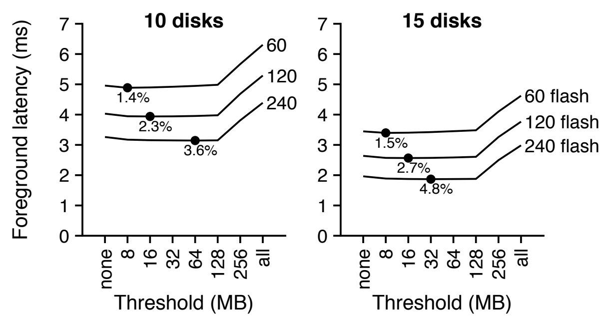

Figure 14: **Flash Buffer.** *We measure how different file-buffering policies impact foreground requests with two plots (for 10 or 15 disks) and three lines (60, 120, or 240GB of flash). Different points on the x-axis represent different policies. The optimum point on each line is marked, showing improvement relative to the latency when no buffering is done.*

5.3 Flash as Buffer

Another advantage of flash is that (due to persistence) it has the potential to reduce disk writes as well as reads. We saw earlier (§4.3) that files tend to be either small and short-lived or big and long-lived, so one strategy would be to store small files in flash and big files on disk.

HDFS writes are considered durable once the data is in memory on every DataNode (but not necessarily on disk), so buffering in flash would not actually improve HDFS write performance. However, decreasing disk writes by buffering the output of *background activities* (*e.g.*, flushes and compaction) indirectly improves *foreground* performance. Foreground activity includes any local requests which could block an HBase request (*e.g.*,

HW	Cost	Failure rate	Performance
HDD	$100/disk	4% AFR [9]	10ms/seek, 100MB/s
RAM	$5.0/GB	4% AFR (8GB)	0 latency
Flash	$0.8/GB	10K P/E cycles	0.5ms latency

Table 3: **Cost Model.** *Our assumptions about hardware costs, failure rates, and performance are presented. For disk and RAM, we state an AFR (annual failure rate), assuming uniform-random failure each year. For flash, we base replacement on wear and state program/erase cycles.*

a `get`). Reducing background I/O means foreground reads will face less competition for disk time. Thus, we measure how buffering files written by background activities affects foreground latencies.

Of course, using flash as a write buffer has a cost, namely less space for caching hot data. We evaluate this tradeoff by measuring performance when using flash to buffer only files which are beneath a certain size. Figure 14 shows how latency corresponds to the policy. At the left of the x-axis, writes are never buffered in flash, and at the right of the x-axis, all writes are buffered. Other x-values represent thresholds; only files smaller than the threshold are buffered. The plots show that buffering all or most of the files results in very poor performance. Below 128MB, though, the choice of how much to buffer makes little difference. The best gain is just a 4.8% reduction in average latency relative to performance when no writes are buffered.

Conclusion: Using flash to buffer all writes results in much worse performance than using flash only as a cache. If flash is used for both caching and buffering, and if policies are tuned to only buffer files of the right size, then performance can be slightly improved. We conclude that these small gains are probably not worth the added complexity, so flash should be for caching only.

5.4 Is Flash worth the Money?

Adding flash to a system can, if used properly, only improve performance, so the interesting question is, given that we want to buy performance with money, *should we buy flash, or something else?* We approach this question by making assumptions about how fast and expensive different storage mediums are, as summarized in Table 3. We also state assumptions about component failure rates, allowing us to estimate operating expenditure.

We evaluate 36 systems, with three levels of RAM (10GB, 30GB, or 100GB), four levels of flash (none, 60GB, 120GB, or 240GB), and three levels of disk (10, 15, or 20 disks). Flash and RAM are used as a hybrid cache with the promote policy (§5.2). For each system, we compute the capex (capital expenditure) to initially purchase the hardware and determine via simulation the foreground latencies (defined in §5.3). Figure 15 shows the cost/performance of each system. 11 of the systems

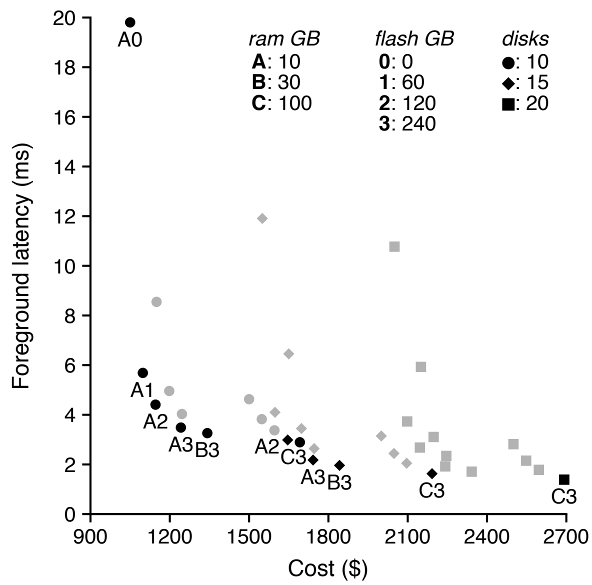

Figure 15: **Capex/latency tradeoff.** *We present the cost and performance of 36 systems, representing every combination of three RAM levels, four flash levels, and three disk levels. Combinations which present unique tradeoffs are black and labeled; unjustifiable systems are gray and unlabeled.*

(31%) are highlighted; these are the only systems that one could justify buying. Each of the other 25 systems is both slower and more expensive than one of these 11 justifiable systems. Over half of the justifiable systems have maximum flash. It is worth noting that the systems with less flash are justified by low cost, not good performance. With one exception (15-disk A2), all systems with less than the maximum flash have the minimum number of disks and RAM. We observe that flash can greatly improve performance at very little cost. For example, A1 has a 60GB SSD but is otherwise the same as A0. With 10 disks, A1 costs only 4.5% more but is 3.5x faster. We conclude that if performance is to be bought, then (within the space we explore) flash should be purchased first.

We also consider expected opex (operating expenditure) for replacing hardware as it fails, and find that replacing hardware is relatively inexpensive compared to capex (not shown). Of the 36 systems, opex is at most $90/year/machine (for the 20-disk C3 system). Furthermore, opex is never more than 5% of capex. For each of the justifiable flash-based systems shown in Figure 15, we also do simulations using KP for flash hits. KP decreased opex by 4-23% for all flash machines while increasing latencies by 2-11%. However, because opex is low in general, the savings are at most $14/year/machine.

Conclusion: Not only does adding a flash tier to the FM stack greatly improve performance, but it is the most cost-effective way of improving performance. In some cases, adding a small SSD can triple performance while only increasing monetary costs by 5%.

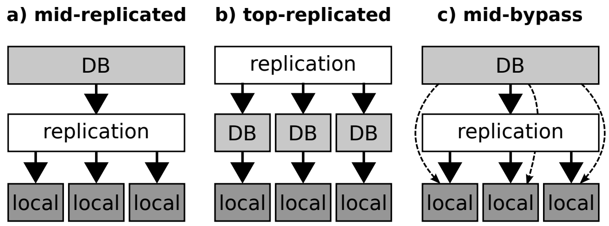

Figure 16: **Layered architectures.** *The HBase architecture (mid-replicated) is shown, as well as two alternatives. Top-replication reduces network I/O by co-locating database computation with database data. The mid-bypass architecture is similar to mid-replication, but provides a mechanism for bypassing the replication layer for efficiency.*

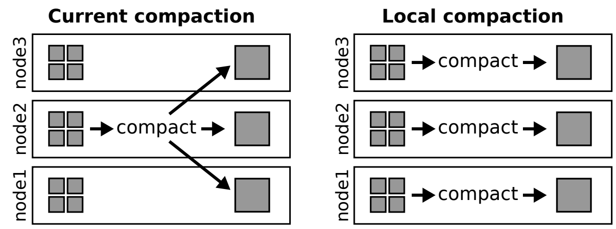

Figure 17: **Local-compaction architecture.** *The HBase architecture (left) shows how compaction currently creates a data flow with significant network I/O, represented by the two lines crossing machine boundaries. An alternative (right) shows how local reads could replace network I/O*

6 Layering: Pitfalls and Solutions

The FM stack, like most storage, is a composition of other systems and subsystems. Some composition is horizontal; for example, FM stores small data in HBase and large data in Haystack (§4.1). In this section, we focus instead on the vertical composition of layers, a pattern commonly used to manage and reduce software complexity. We discuss different ways to organize storage layers (§6.1), how to reduce network I/O by bypassing the replication layer (§6.2), and how to reduce the randomness of disk I/O by adding special HDFS support for HBase logging (§6.3).

6.1 Layering Background

Three important layers are the *local layer* (*e.g.*, disks, local file systems, and a DataNode), the *replication layer* (*e.g.*, HDFS), and the *database layer* (*e.g.*, HBase). FM composes these in a *mid-replicated* pattern (Figure 16a), with the database at the top of the stack and the local stores at the bottom. The merit of this architecture is simplicity. The database can be built with the assumption that underlying storage, because it is replicated, will be available and never lose data. The replication layer is also relatively simple, as it deals with data in its simplest form (*i.e.*, large blocks of opaque data). Unfortunately, mid-replicated architectures separate computation from data. Computation (*e.g.*, database operations such as compaction) can only be co-resident with at most one replica, so all writes involve network transfers.

Top-replication (Figure 16b) is an alternative approach used by the Salus storage system [29]. Salus supports the standard HBase API, but its top-replicated approach provides additional robustness and performance advantages. Salus protects against memory corruption and certain bugs in the database layer by replicating database computation as well as the data itself. Doing replication above the database level also reduces network I/O. If the database wants to reorganize data on disk (*e.g.*, via compaction), each database replica can do so on its local copy. Unfortunately, top-replicated storage is complex. The database layer must handle underlying failures as well as cooperate with other databases; in Salus, this is accomplished with a pipelined-commit protocol and Merkle trees for maintaining consistency.

Mid-bypass (Figure 16c) is a third option proposed by Zaharia *et al.* [30]. This approach (like mid-replication), places the replication layer between the database and the local store, but in order to improve performance, an *RDD* (Resilient Distributed Dataset) API lets the database bypass the replication layer. Network I/O is avoided by shipping computation directly to the data. HBase compaction could be built upon two RDD transformations, *join* and *sort*, and network I/O could thus be avoided.

6.2 Local Compaction

We simulate the mid-bypass approach, with compaction operations shipped directly to all the replicas of compaction inputs. Figure 17 shows how local compaction differs from traditional compaction; network I/O is traded for local I/O, to be served by local caches or disks.

Figure 18 shows the result: a 62% reduction in network reads from 3.5TB to 1.3TB. The figure also shows disk reads, with and without local compaction, and with either write allocate (wa) or no-write allocate (nwa) caching policies (§5.1). We observe disk I/O increases sightly more than network I/O decreases. For example, with a 100GB cache, network I/O is decreased by 2.2GB but disk reads are increased by 2.6GB for no-write allocate. This is unsurprising: HBase uses secondary replicas for fault tolerance rather than for reads, so secondary replicas are written once (by a flush or compaction) and read at most once (by compaction). Thus, local-compaction reads tend to (a) be misses and (b) pollute the cache with data that will not be read again. We see that write allocate still underperforms no-write allo-

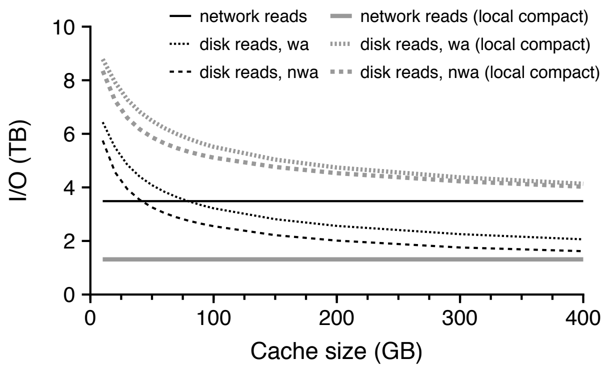

Figure 18: **Local-compaction results.** *The thick gray lines represent HBase with local compaction, and the thin black lines represent HBase currently. The solid lines represent network reads, and the dashed lines represent disk reads; long-dash represents the no-write allocate cache policy and short-dash represents write allocate.*

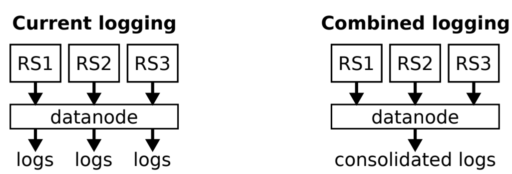

Figure 19: **Combined-logging architecture.** *Currently (left), the average DataNode will receive logs from three HBase RegionServers, and these logs will be written to different locations. An alternative approach (right) would be for HDFS to provide a special logging API which allows all the logs to be combined so that disk seeks are reduced.*

cate (§5.1). However, write allocate is now somewhat more competitive for large cache sizes because it is able to serve some of the data read by local compaction.

Conclusion: Doing local compaction by bypassing the replication layer turns over half the network I/O into disk reads. This is a good tradeoff as network I/O is generally more expensive than sequential disk I/O.

6.3 Combined Logging

We now consider the interaction between replication and HBase logging. Figure 19 shows how (currently) a typical DataNode will receive log writes from three RegionServers (because each RegionServer replicates its logs to three DataNodes). These logs are currently written to three different local files, causing seeks. Such seeking could be reduced if HDFS were to expose a special logging feature that merges all logical logs into a single physical log on a dedicated disk as illustrated.

We simulate combined logging and measure performance for requests which go to disk; we consider latencies for foreground reads (defined in §5.1), compaction, and logging. Figure 20 reports the results for varying numbers of disks. The latency of log writes decreases dramatically with combined logging; for example, with 15 disks, the latency is decreased by a factor of six. Compaction requests also experience modest gains due to less competition for disk seeks. Currently, neither logging nor compaction block the end user, so we also consider the performance of foreground reads. For this metric, the gains are small, *e.g.*, latency only decreases by 3.4% with 15 disks. With just 10 disks, dedicating one disk to logging slightly hurts user reads.

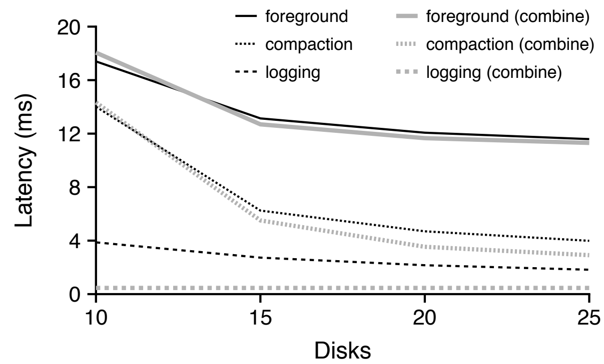

Figure 20: **Combined logging results.** *Disk latencies for various activities are shown, with (gray) and without (black) combined logging.*

Conclusion: Merging multiple HBase logs on a dedicated disk reduces logging latencies by a factor of 6. However, `put` requests do not currently block until data is flushed to disks, and the performance impact on foreground reads is negligible. Thus, the additional complexity of combined logging is likely not worthwhile given the current durability guarantees. However, combined logging could enable HBase, at little performance cost, to give the additional guarantee that data is on disk before a `put` returns. Providing such a guarantee would make logging a foreground activity.

7 Related Work

In this work, we compare the I/O patterns of FM to prior GFS and HDFS workloads. Chen *et al.*[4] provides broad characterizations of a wide variety of MapReduce workloads, making some of the comparisons possible. The MapReduce study is *broad*, analyzing traces of coarse-grained events (*e.g.*, file opens) from over 5000 machines across seven clusters. By contrast, our study is *deep*, analyzing traces of fine-grained events (*e.g.*, reads to a byte) for just nine machines.

Detailed trace analysis has also been done in many non-HDFS contexts, such as the work by Baker *et al.* [2]

in a BSD environment and by Harter *et al.* [13] for Apple desktop applications. Other studies include the work done by Ousterhout *et al.* [21] and Vogels *et al.* [28].

A recent photo-caching study by Huang *et al.* [15] focuses, much like our work, on I/O patterns across multiple layers of the stack. The photo-caching study correlated I/O across levels by tracing at each layer, whereas our approach was to trace at a single layer and infer I/O at each underlying layer via simulation. There is a tradeoff between these two methodologies: tracing multiple levels avoids potential inaccuracies due to simulator oversimplifications, but the simulation approach enables greater experimentation with alternative architectures beneath the traced layer.

Our methodology of trace-driven analysis and simulation is inspired by Kaushik *et al.* [16], a study of Hadoop traces from Yahoo! Both the Yahoo! study and our work involved collecting traces, doing analysis to discover potential improvements, and running simulations to evaluate those improvements.

We are not the first to suggest the methods we evaluated for better HDFS integration (§6); our contribution is to quantify how useful these techniques are for the FM workload. The observation that doing compaction above the replication layer wastes network bandwidth has been made by Wang *et al.* [29], and the approach of local compaction is a specific application of the more general techniques described by Zaharia *et al.* [30]. Combined logging is also commonly used by administrators of traditional databases [8, 22].

8 Conclusions

We have presented a detailed multilayer study of storage I/O for Facebook Messages. Our combined approach of analysis and simulation allowed us to identify potentially useful changes and then evaluate those changes. We have four major conclusions.

First, the special handling received by writes make them surprisingly expensive. At the HDFS level, the read/write ratio is 99/1, excluding HBase compaction and logging overheads. At the disk level, the ratio is write-dominated at 36/64. Logging, compaction, replication, and caching all combine to produce this write blowup. Thus, optimizing writes is very important even for especially read-heavy workloads such as FM.

Second, the GFS-style architecture is based on workload assumptions such as "high sustained bandwidth is more important than low latency" [11]. For FM, many of these assumptions no longer hold. For example, we demonstrate (§5.1) just the opposite is true for FM: because I/O is highly random, bandwidth matters little, but latency is crucial. Similarly, files were assumed to be very large, in the hundreds or thousands of megabytes. This traditional workload implies a high data-to-metadata ratio, justifying the one-NameNode design of GFS and HDFS. By contrast, FM is dominated by small files; perhaps the single-NameNode design should be revisited.

Third, FM storage is built upon layers of independent subsystems. This architecture has the benefit of simplicity; for example, because HBase stores data in a replicated store, it can focus on high-level database logic instead of dealing with dying disks and other types of failure. Layering is also known to improve reliability, *e.g.*, Dijkstra found layering "proved to be vital for the verification and logical soundness" of an OS [7]. Unfortunately, we find that the benefits of simple layering are not free. In particular, we showed (§6) that building a database over a replication layer causes additional network I/O and increases workload randomness at the disk layer. Fortunately, simple mechanisms for sometimes bypassing replication can reduce layering costs.

Fourth, the cost of flash has fallen greatly, prompting Gray's proclamation that "tape is dead, disk is tape, flash is disk" [12]. To the contrary, we find that for FM, flash is not a suitable replacement for disk. In particular, the cold data is too large to fit well in flash (§4.1) and the hot data is too large to fit well in RAM (§5.1). However, our evaluations show that architectures with a small flash tier have a positive cost/performance tradeoff compared to systems built on disk and RAM alone.

In this work, we take a unique view of Facebook Messages, not as a single system, but as a complex composition of systems and subsystems, residing side-by-side and layered one upon another. We believe this perspective is key to deeply understanding modern storage systems. Such understanding, we hope, will help us better integrate layers, thereby maintaining simplicity while achieving new levels of performance.

9 Acknowledgements

We thank the anonymous reviewers and Andrew Warfield (our shepherd) for their tremendous feedback, as well as members of our research group for their thoughts and comments on this work at various stages. We also thank Pritam Damania, Adela Maznikar, and Rishit Shroff for their help in collecting HDFS traces.

This material was supported by funding from NSF grants CNS-1319405 and CNS-1218405 as well as generous donations from EMC, Facebook, Fusion-io, Google, Huawei, Microsoft, NetApp, Sony, and VMware. Tyler Harter is supported by the NSF Fellowship and Facebook Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or other institutions.

References

[1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2014.

[2] Mary Baker, John Hartman, Martin Kupfer, Ken Shirriff, and John Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 198–212, Pacific Grove, California, October 1991.

[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 205–218, Seattle, Washington, November 2006.

[4] Chen, Yanpei and Alspaugh, Sara and Katz, Randy. Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads. *Proc. VLDB Endow.*, August 2012.

[5] Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Journal-guided Resynchronization for Software RAID. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, pages 87–100, San Francisco, California, December 2005.

[6] Dhruba Borthakur and Kannan Muthukkaruppan and Karthik Ranganathan and Samuel Rash and Joydeep Sen Sarma and Nicolas Spiegelberg and Dmytro Molkov and Rodrigo Schmidt and Jonathan Gray and Hairong Kuang and Aravind Menon and Amitanand Aiyer. Apache Hadoop Goes Realtime at Facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*, Athens, Greece, June 2011.

[7] E. W. Dijkstra. The Structure of the THE Multiprogramming System. *Communications of the ACM*, 11(5):341–346, May 1968.

[8] IBM Product Documentation. Notes/domino best practices: Transaction logging. http://www-01.ibm.com/support/docview.wss?uid=swg27009309, 2013.

[9] Ford, Daniel and Labelle, François and Popovici, Florentina I. and Stokely, Murray and Truong, Van-Anh and Barroso, Luiz and Grimes, Carrie and Quinlan, Sean. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, December 2010.

[10] Gregory R. Ganger. Blurring the Line Between Oses and Storage Devices. Technical Report CMU-CS-01-166, Carnegie Mellon University, December 2001.

[11] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing, New York, October 2003.

[12] Jim Gray. Tape is Dead. Disk is Tape. Flash is Disk, RAM Locality is King, 2006.

[13] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[14] Joseph L. Hellerstein. Google cluster data. Google research blog, January 2010. Posted at http://googleresearch.blogspot.com/2010/01/google-cluster-data.html.

[15] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An Analysis of Facebook Photo Caching. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 167–181, Farmington, Pennsylvania, November 2013.

[16] Rini T. Kaushik and Milind A Bhandarkar. GreenHDFS: Towards an Energy-Conserving, Storage-Efficient, Hybrid Hadoop Compute Cluster. In *The 2010 Workshop on Power Aware Computing and Systems (HotPower '10)*, Vancouver, Canada, October 2010.

[17] Niall Kennedy. Facebook's Photo Storage Rewrite. http://www.niallkennedy.com/blog/2009/04/facebook-haystack.html, April 2009.

[18] Butler W. Lampson. Hints for Computer System Design. In *Proceedings of the 9th ACM Symposium on Operating System Principles (SOSP '83)*, pages 33–48, Bretton Woods, New Hampshire, October 1983.

[19] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, Cambridge, Massachusetts, October 1996.

[20] Kannan Muthukkaruppan. Storage Infrastructure Behind Facebook Messages. In *Proceedings of International Workshop on High Performance Transaction Systems (HPTS '11)*, Pacific Grove, California, October 2011.

[21] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the 10th ACM Symposium on Operating System Principles (SOSP '85)*, pages 15–24, Orcas Island, Washington, December 1985.

[22] Matt Perdeck. Speeding up database access. http://www.codeproject.com/Articles/296523/Speeding-up-database-access-part-8-Fixing-memory-d, 2011.

[23] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[24] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST '10)*, Incline Village, Nevada, May 2010.

[25] Jason Sobel. Needle in a haystack: Efficient storage of billions of photos. http://www.flowgram.com/p/2qi3k8eicrfgkv, June 2008.

[26] Nicolas Spiegelberg. Allow record compression for hlogs. https://issues.apache.org/jira/browse/HBASE-8155, 2013.

[27] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 224–237, Saint-Malo, France, October 1997.

[28] Werner Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 93–109, Kiawah Island Resort, South Carolina, December 1999.

[29] Yang Wang and Manos Kapritsos and Zuocheng Ren and Prince Mahajan and Jeevitha Kirubanandam and Lorenzo Alvisi and Mike Dahlin. Robustness in the Salus Scalable Block Store. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation*, Lombard, Illinois, April 2013.

[30] Zaharia, Matei and Chowdhury, Mosharaf and Das, Tathagata and Dave, Ankur and Ma, Justin and McCauley, Murphy and Franklin, Michael J. and Shenker, Scott and Stoica, Ion. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, San Jose, California, April 2010.