How to Teach an Old File System Dog New Object Store Tricks

Eunji Lee[†], Youil Han[†], Suli Yang[‡], Andrea C. Arpaci-Dusseau[‡], Remzi H. Arpaci-Dusseau[‡]
[†]*Chungbuk National University,* [‡]*University of Wisconsin–Madison*

Abstract

Many data service platforms use local file systems as their backend storage. Although this approach offers advantages in portability, extensibility, and ease of development, it may suffer from severe performance degradation if the mapping between the services required by the data service platform and the functions provided by the local file system is not carefully managed. This paper presents in-depth analysis of performance problems in current data service platforms that use file systems as their backend storage and proposes three novel strategies that are essential to solving the current performance problems. We demonstrate the efficacy of our strategies by implementing a prototype object store in Ceph, called SwimStore (Shadowing with Immutable Metadata Store). We experimentally show that SwimStore provides high performance with little variation, as well as a large reduction in write traffic.

1 Introduction

We are witnessing an explosive growth of digital data both in volume and variety. To cope with such rapidly growing unstructured data, there has been a surge of interest in data service platforms such as distributed NoSQL systems like BigTable [2], Cassandra [5], HBase [8], MongoDB [17], Ceph [32], Swift [1], distributed file systems like GFS [6] and HDFS [25], and embedded key-value stores like LevelDB [7], HyperLevelDB [10], and RocksDB [3].

Most data service platforms use a layered approach rather than building the platform end-to-end. For example, BigTable [2] and HBase [8] rely on a distributed file system underneath, which is subsequently backed by a local file system. Cassandra [5] and MongoDB [17] use a per-node storage engine that is typically running atop a file system. Embedded key-value stores such as LevelDB [7] and RocksDB [3] employ a local file system as their storage backend. The fundamental rationale behind this layered architecture is *abstraction*. Abstracting away underlying details, the layered approach permits ease of development and maintenance in building a complicated data service platform.

A layered approach, however, sometimes backfires when the lower layer was not originally designed for the upper layer; that is the case between many data service platforms and the local file system underneath. Most of file systems today and their POSIX standard interface were not intended to serve as a local storage engine for other data service platforms, so as it has been pointed out in [31], there is a mismatch between the two, both inside and outside.

Inside, local file systems are optimized for system-wide performance and reliability, largely ignoring demands from individual applications. Thus, the upper layer applications have very little control over what's going on inside the file system. As a result, the upper layer can suffer from degraded QoS (Quality-of-Service) resulting from unintended and unexpected *entanglement* [14] within the file system. Such entanglement arises, for example, from periodic flushing of dirty blocks and journaling for consistent metadata update. Both techniques aim to fulfill the file system's original objectives (the former for enhancing system-wide performance and the latter for improving reliability), but they can potentially lead to serious performance fluctuations, which greatly degrade the QoS.

Outside, local file systems do not provide support for atomic update, which is a crucial functionality required from the underlying storage backend in data service platforms.[1] This functional mismatch forces data service platforms to use write-ahead logging [16, 22], resulting in significant increase in write traffic due to double writes (one to the log and the other to home locations)[2]. In addition to the traffic increase, the double writes adversely affect the QoS due to periodic flushing of dirty blocks, one form of entanglement in the file system.

To address the performance problems arising from the mismatch, two approaches have been actively pursued in recent years. The first approach, as exemplified by BlueStore [31], an object store for Ceph, is to bypass the file system and access the raw block device directly. This bare-bones approach has a performance advantage over the one with a file system as an intermediary. However, it suffers from the loss of numerous advantages of the file system abstraction such as portability, extensibility, and

[1] The `O_ATOMIC` flag support is underway in XFS, but it is neither yet supported in the mainstream kernel nor considered in other file systems [9, 29].

[2] Many file systems use journaling, a form of write-ahead logging, internally but its purpose is for guaranteeing consistency of the file system's data and metadata, not for supporting an interface for an atomic write [15, 23, 27, 28].

Table 1: Summary of Data Service Platforms.

Type	Platform	Backend Structure within a File System	Data Integrity Support	Target Environment
Key-value store	RocksDB, LevelDB, HyperLevelDB	LSM Tree	Logging	Local
	Dynamo	LSM Tree	Logging	Distributed
Document store	MongoDB, CouchDB	LSM Tree	Logging	Distributed
Column store	HBase, BigTable, Cassandra	LSM Tree	Logging	Distributed
Object store	Ceph, Swift	Store item as a File	Logging	Distributed
Distr. File System	GFS, HDFS	Store item as a File	Logging	Distributed

ease of development.

The second approach is to add new features such as atomic update to the file system interface [26, 33, 9]. This approach might remedy the performance problem, but results in a loss of generality by optimizing the file system for specific workloads.

Our approach is different; we use the file system and the POSIX standard interface as they are, but use files in a manner that minimizes the inefficiency of layering and takes better control over the performance. To this end, we first analyze the current data service platforms in terms of the performance and controllability, particularly correlating them with the underlying file system workings. Then, we present a set of strategies to store unstructured data items over a file system achieving higher performance and QoS. The first strategy is to use files as *metadata immutable containers* for storing data items much like a virtual disk file that stores the contents of a virtual machine's hard disk drive [30]. This approach eliminates not only most of metadata update in the file system but also, and more importantly, write traffic fluctuations arising from forcing the metadata, which is needed to maintain the file system consistency. We also use multiple container files of different allocation unit sizes to minimize the complexity arising from fragmentation.

Second, we use *in-file shadow paging* to eliminate the double write problem associated with the write-ahead logging approach. Although we use shadow paging, we do not have to address the accompanying garbage collection issue since the underlying file system provides a level of indirection and thus the garbage collection can easily be performed by deallocating blocks associated with the *garbage* using the hole punching feature [12] supported by most modern file systems. Also, we make use of intent logging to implement a transaction where data update by shadow paging is one type of record, which points to the target area of update. Finally, we use a variation of the transactional checksum [19] to remove the ordering between the data update and log write.

As a proof of concept, we implement a prototype object store for Ceph called SwimStore (ShadoWing with Immutable Metadata Store) built atop a local file system The measurement study shows that SwimStore improves the performance by twofold, reducing the write traffic by more than a half, compared to other object stores in Ceph such as FileStore and KStore. Even when compared with BlueStore, which bypasses the file system and accesses the raw block device directly, SwimStore performs competitively while retaining all the benefits of the file system abstraction.

2 Overview of Data Service Platforms

The data service platforms mostly rely on a local file system for their final backend storage, while a raw block device is accessed directly on some occasions. This trend raises one central question: what is the best way to store unstructured data atop a local file system? To find an answer to this question, we classify data service platforms into two types, according to how they store data items over a local file system, and investigate strengths and weaknesses of each type. Table 1 summarizes the key features of data service platforms.

Packing: This architecture maintains multiple items in a single file. The most common instance of this architecture is the log-structured merge tree (LSM tree), which is extensively used as in-file data structures in numerous key-value stores [3, 7, 10, 34, 13]. The LSM tree buffers data items in memory and batches them into a file when the amount of written data reaches the limit [21]. The data items are maintained in sorted order by key, managed by a skip list in memory [20] and by a sorted-string table (sstable) within a file. The multiple sstables are periodically merged into fewer ones for better read performance and space reclaiming, which is called *compaction*. The LSM tree has the advantage of using the full bandwidth of the underlying storage, translating small sized updates into a sequential write stream [18]. Moreover, keeping data in sorted order makes it possible to retrieve data with O(log(n)) complexity and to process a range query quickly without any additional indexing.

However, the LSM tree originally targets the small data items, and thus it does not perform well for large writes. When a write is large, the memory buffer fills up quickly, forcing a frequent flush of buffered data. This activity results in a large number of sstables that con-

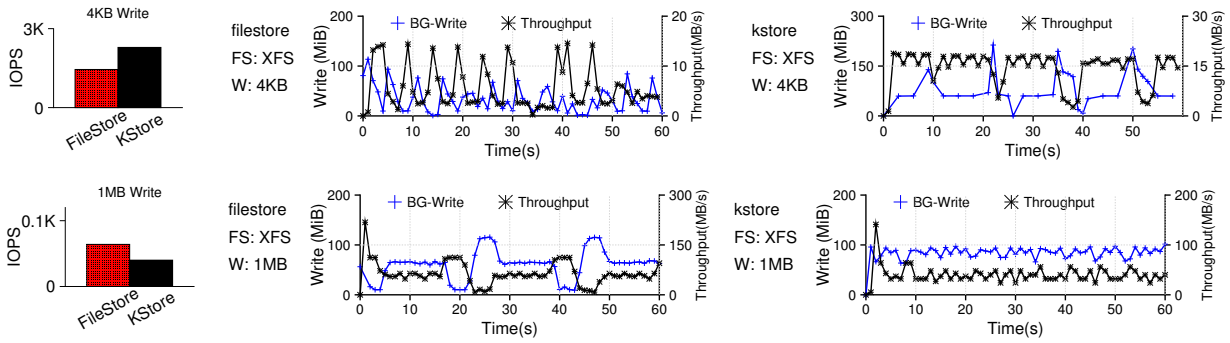


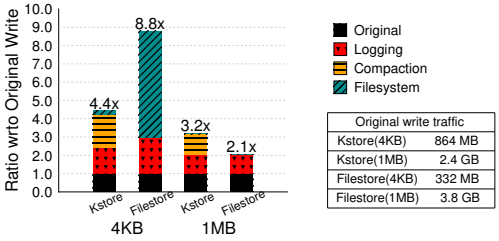
Figure 1: **Average IOPS and the Performance Change over Time.** *We run 16 threads each issuing 4KB and 1MB object writes for 60s. The leftmost figure shows that KStore provides 1.58x higher average IOPS than FileStore for small writes, while FileStore outperforms KStore by 2.3x for large writes. The rest of figures show the throughput of two object stores over time (indicated by a black line), in conjunction of the write traffic caused by the background activities, such as a periodic flush of a file system and a compaction operation in key-value stores.*

tains only a few items but also have ranges that overlap with others. In this situation, the cost of data retrieval increases significantly because a system has to search multiple sstables that have a range encompassing the target key. To avoid this weakness, the compaction must be performed at short intervals so as to make a globally sorted sequence of data items. Unfortunately, the write amplification by compaction is exceedingly destructive when the write size is large because the performance is dictated by the total amount of write traffic.

Furthermore, in the packing architecture, the partial update of data items is challenging. Thus, LSM tree based key-value stores handle such operations (e.g., `append`, `update`, or `truncate`) by inserting an updated data item as a whole, which is particularly expensive for large data.

Mapping: This architecture associates a data item with a file, and it is mainly used in distributed object stores (e.g., Ceph, Swift) and distributed file systems (e.g., GFS, HDFS). As opposed to key-value stores, these systems are designed to target on large items. HDFS processes data on a block granularity whose default size is 64MB. Ceph's object store was originally meant to be a backend for a distributed file system in which the objects are multiple MBs. The primary benefit of this approach lies in the increased developer velocity. By associating a data item with a file, we can rely on a number of great functionalities of a file system, such as indexing, caching, and space management, instead of developing them from scratch. Furthermore, as most of these mechanisms have been carefully designed and heavily tested over past decades, their use can help to improve system reliability. Also, the partial manipulation of items becomes easier because the file systems already support such functions for file components.

However, this approach also has a downside. First, this design, which is tightly coupled with the file system



Original write traffic	
Kstore(4KB)	864 MB
Kstore(1MB)	2.4 GB
Filestore(4KB)	332 MB
Filestore(1MB)	3.8 GB

Figure 2: **Write Traffic Break-down.**

causes the data service platform to be greatly affected by file system internal activity. As a result, data service platforms might confront unexpected delay or performance oscillation incurred by out-of-control workings of the underlying file systems.

In addition, the high dependency on the file system facilities implies that the implementation efficiency can be severely constrained by the limited POSIX APIs. For example, maintaining additional metadata for data items is not easy when those items are stored as a file because conventional file systems allow only a limited number of attributes for a file, and these limitations even vary across file systems.

Besides this, the absence of required operations such as atomic updates and efficient file grouping / splitting, and file enumeration in sorted order lead us to suffer from sub-optimal performance in many cases. Finally, the file system metadata can be a burden in the presence of a myriad of files, amplifying write traffic greatly, particularly for small data items.

3 Quantitative Analysis

We perform a quantitative analysis on current backend storage architectures, particularly focusing on the correlation between their performance and the underlying file system behaviors.

Experimental Setup: Our empirical study is performed using Ceph [32], a distributed object store platform, be-

cause it enables the fair comparison of two architectures by offering an option for choosing backend storage between packing (KStore) and mapping (FileStore) architectures [11]. KStore is an object store provided by Ceph that manages data using embedded key-value store (i.e., RocksDB), representing the packing architecture. FileStore, which is also in Ceph, stores each object as a file, representing the mapping architecture. By comparing these object stores, we examine how different backend storage architectures of the NoSQL systems work in general for various environments.

All our experiments are performed on Amazon EC2 clusters; each machine consists of Intel Xeon quad-core running at 2.5GHz, 16GB DDR, and two 256 GiB SSDs. We use Ubuntu Server 16.04 and XFS file system.

Performance: We investigate the performance of different backend architectures with respect to the object size by measuring the IOPS of KStore (packing) and FileStore (mapping) writing 4KB and 1MB objects. Figure 1 shows that KStore provides 1.57x higher IOPS than FileStore for small writes, while FileStore outperforms KStore by 1.6x for large writes. We attribute the low performance of FileStore in small writes to the severe metadata overhead coming from file creation and indexing, which is mostly offset in large writes. On contrary, the LSM tree based packing architecture turns out to be a poor fit for handling large objects, supposedly due to the frequent compaction and resultant write amplification.

Write Amplification: To validate our reasoning, we measure the total write traffic for each scenario and break down it according to where it comes from. Figure 2 shows FileStore amplifies write traffic by 8.8x for small writes, leading to the huge write amplification by a file system. When the objects are large, KStore yields larger write amplification than FileStore, paying the additional cost for compaction. Note that FileStore has no compaction because it does not keep objects sorted, depending on file system facilities for object retrieval or traversal. In both object stores, the double-write penalty is observed which arises from write-ahead logging.

Controllability: Figure 1 shows the performance change over time when writing small and large objects for 60s. We can see that FileStore suffers from the periodic performance fluctuation both in small and large writes. In this case, the performance jitter results from the periodic bulky writes invoked by the background file system activities. Specifically, FileStore uses buffered I/O by default, in which updates are buffered in a page cache and batched to storage by a periodic flush daemon or a journaling commit process. This mechanism increases overall I/O performance, but it is detrimental for QoS because it can slow down a system at untimely point.

KStore manifests with different patterns of performance spikes. Our detailed analysis reveals that the cause of performance fluctuation here lies in the user-level activity (i.e., compaction), not a file system. Because RocksDB (operating in KStore) holds data in a user-level buffer, instead of making use of the system-level page cache, it can avoid the unexpected delay rooted from a page cache. However, without careful consideration, this approach can suffer from a more serious performance problem. For an example, as illustrated in Figure 1, KStore provides a poor performance in large writes, because the frequent compaction of RocksDB decreases I/O bandwidth significantly.

4 SwimStore

In the previous section, we examined the pros and cons of two different backend architectures that run atop a local file system. In this section, we present SwimStore, a new backend storage architecture that pursues the following goals: (1) providing excellent performance for various sizes of data, (2) taming performance oscillation by minimizing out-of-control background activities (e.g., compaction, periodic flush) (3) reducing write amplification by eliminating the logging and metadata overhead. We achieve these goals with three key strategies.

Metadata Immutable Container: Metadata updates are a main source of performance degradation when using a file system as a backend storage. They are particularly problematic when the small data is stored as a file, like in FileStore. Creating a new file, although conceptually simple, involves the update of numerous types of metadata and incurs significant overheads. To avoid this overhead, we use a file as a container for multiple data items thereby amortizing its creation overheads over them. Furthermore, we make the container *metadata immutable* meaning that the container does not incur the file metadata manipulation when storing objects within it. This concept is realized by creating a group of preallocated files at deployment, eliminating all of the runtime metadata updates associated with file creations and space allocation.

In-file Shadow Paging: Another problem of using a file system as a storage backend is double writes resulting from write-ahead logging. This behavior can be more harmful in large writes where the performance is determined by the total I/O traffic. Moreover, the writes to the file system after logging are performed by buffered I/Os, which introduce fluctuations in write traffic by periodic flushings. In SwimStore, we use a shadow paging technique where data update is made in an out-of-place manner and the space taken up by old data is recycled after the update is persisted [22]. In SwimStore, the out-of-place update is made to a free area in the container file with the `O_DIRECT` and `O_DSYNC` flags set. This syn-

chronous update may increase the response time but its delay can be made comparable to accessing the raw device directly by pre-allocating the file data blocks in advance. Indeed, the shadowing technique is hard to use in conjunction with the LSM tree, because its synchronous and out-of-place write does not allow items to be sorted in a memory buffer, forcing every write to storage.

Note that a write request, which needs to be handled with atomicity in the object store, comprises a series of sub-operations involving updates of multiple data and metadata [4]. To make all updates take place by a primitive action, we use a small intent log that records the location of target data (e.g., file name, offset, and length), and the associated metadata. In the intent log, we record a checksum over not only the data written but also including all the log records of the transaction, thereby safely recovering from a crash without ordering constraints.

Slotted Allocation with Hole-punching: In shadow paging, after a data is updated, the old copy becomes garbage and needs to be recycled to make free space. The free space management in general and garbage collection in particular are one of the most difficult problems in storage management systems [24]. Our approach in this paper is to use multiple container files whose allocation unit sizes are recursively doubled. For example, the first container file may have an allocation unit size of 4 KiB, the second 8 KiB, etc. With this setting, when new data is written, its space allocation is made by the container file whose allocation unit size is best fit. Also, the whole unit is allocated to avoid the need to manage fragments. Similarly, when the allocated space is recycled, the whole allocation unit is returned.

This space management enables to avoid external fragmentation, but it impedes the aggregation of multiple writes, which is particularly useful optimization for highly concurrent workloads. To hold onto the benefits of aggregated I/O, SwimStore makes use of the hole punching feature [12] supported by most modern file systems along with slotted allocation. The punch-hole function deallocates physical space associated with the file range, preventing space waste by the file data not in use. By using this feature for bursty writes appropriately, we can achieve the high performance and space efficiency simultaneously.

5 Performance Evaluation

The prototype of SwimStore is implemented with 12K LOC in Ceph 12.01 running on Linux 4.4.0. We measure the performance on the same experimental platforms as described in § 3. We create ten container files at the deployment time whose allocation sizes are recursively doubled (4KB to 2MB). The data blocks of container files are pre-allocated by means of `fallocate` system call. It is noteworthy that Ceph maintains the meta-

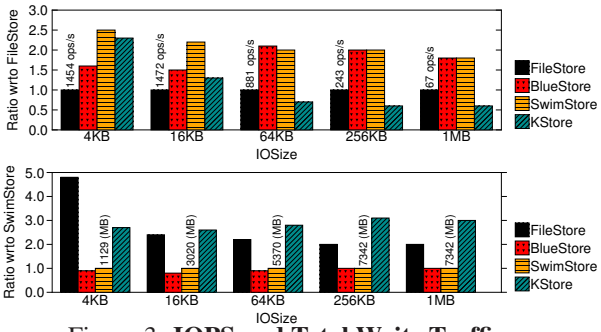


Figure 3: **IOPS and Total Write Traffic.**

data for each object, such as object name or timestamp. SwimStore and FileStore both manage the object metadata using the embedded key-value store, which is LevelDB here, because metadata updates are mostly small sized. For performance evaluation, we use the Rados benchmark offered by Ceph [32] which issues write requests to the object store layer, so as to observe their performances in a more general and clear environment.

Figure 3 compares the IOPS of different object stores when varying the object size from 4KB to 1MB. For the more practical analysis, we also study BlueStore, which currently serves as the default backend storage of Ceph, managing a raw device directly bypassing a file system. In the figure, we can observe that SwimStore provides excellent performance for the full range of object sizes. This result is contrast to KStore that delivers high IOPS for small writes, but provides drastically decreased performance for large writes. Compared to FileStore, SwimStore offers a twofold performance for varying object sizes. SwimStore also outperforms FileStore and KStore in terms of the write amplification; FileStore and KStore have upto 4.8x and 3x more writes, respectively, compared to SwimStore. This result demonstrates that our strategies devised to reduce the write traffic in object store is highly effective in a real system.

Figure 3 shows that SwimStore achieves performance comparable to that of BlueStore without the burden of managing the raw block device. SwimStore even outperforms BlueStore for write sizes smaller than 32 KiB. We attribute this performance advantage for small writes to an optimization we perform in SwimStore: wherever possible, we aggregate multiple write operations to improve throughput. We find that this optimization is particularly useful for highly concurrent workloads, which are common in data service platforms.

6 Conclusion

This paper presents SwimStore, a backend storage architecture for data service platforms that achieves high write performance, low write amplification, and little performance variations, running atop a file system. The prototype of SwimStore built within Ceph outperforms FileStore and KStore which also use a file system, and pro-

vides at least competitive performance as compared to BlueStore.

7 Acknowledgments

We thank the anonymous reviewers and ADG members for their insightful comments. This work was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (No. 2017R1D1A1B03031494).

References

[1] APPLE. Swift. https://docs.openstack.org/swift/latest/.

[2] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems 26*, 2 (June 2008), 4:1–4:26.

[3] FACEBOOK. A persistent key-value store. http://rocksdb.org.

[4] FACTOR, M., METH, K., NAOR, D., RODEH, O., AND SATRAN, J. Object storage: The future building block for storage systems. In *Local to Global Data Interoperability-Challenges and Technologies* (2005), IEEE, pp. 119–123.

[5] FOUNDATION, T. A. S. Cassandra. https://github.com/apache/cassandra.

[6] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA, 2003), SOSP '03, ACM, pp. 29–43.

[7] GOOGLE. Leveldb. http://github.com.google/leveldb.

[8] HARTER, T., BORTHAKUR, D., DONG, S., AIYER, A. S., TANG, L., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Analysis of hdfs under hbase: a facebook messages case study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies* (Santa Clara, CA, 2014), FAST '14, USENIX Association, pp. 199–212.

[9] HELLWIG, C. Failure-atomic file updates for linux. In *Linux Piter* (2017), Linux Foundation.

[10] HYPERDEX. Hyperleveldb. https://github.com/rescrv/HyperLevelDB.

[11] LEE, D.-Y., JEONG, K., HAN, S.-H., KIM, J.-S., HWANG, J.-Y., AND CHO, S. Understanding write behaviors of storage backends in ceph object store. In *Proceedings of the 2017 IEEE International Conference on Massive Storage Systems and Technology* (2017), vol. 10.

[12] LINUX. fallocate punch-hole. http://man7.org/linux/man-pages/man2/fallocate.2.html.

[13] LU, L., PILLAI, T. S., GOPALAKRISHNAN, H., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS) 13*, 1 (2017), 5.

[14] LU, L., ZHANG, Y., DO, T., AL-KISWANY, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Physical disentanglement in a container-based file system. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation* (Broomfield, CO, 2014), OSDI '14, USENIX Association, pp. 81–96.

[15] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., TOMAS, A., AND VIVIER, L. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium* (2007), vol. 2, pp. 21–33.

[16] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems 17*, 1 (March 1992), 94–162.

[17] MONGODB, I. Mongodb. https://www.mongodb.com.

[18] ONEIL, P., CHENG, E., GAWLICK, D., AND ONEIL, E. The log-structured merge-tree (lsm-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[19] PRABHAKARAN, V. *IRON File Systems*. PhD thesis, University of Wisconsin, Madison, June 2006.

[20] PUGH, W. A skip list cookbook. In *University of Maryland* (Los Angeles, CA, USA, 1990).

[21] RAJU, P., KADEKODI, R., CHIDAMBARAM, V., AND ABRAHAM, I. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 497–514.

[22] RAMAKRISHNAN, R., AND GEHRKE, J. *Database Management Systems*. McGraw-Hill Computer Science Series. McGraw-Hill, 2000.

[23] REISER, H. Reiserfs. https://www.namesys.com.

[24] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (February 1992), 26–52.

[25] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The hadoop distributed file system. In *Proceedings of the 36th IEEE Symposium on Mass Storage Systems and Technologies* (Incline Village, NV, USA, 2010), MSST '10, IEEE, pp. 1–10.

[26] SPILLANE, R. P., GAIKWAD, S., CHINNI, M., ZADOK, E., AND WRIGHT, C. P. Enabling transactional file access via lightweight kernel extensions. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies* (San Francisco, California, 2009), FAST '09, USENIX Association, pp. 29–42.

[27] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the xfs file system. In *Proceedings of the USENIX 1996 Annual Technical Conference* (San Diego, CA, 1996), ATC '96, USENIX Association.

[28] TWEEDIE, S. C. Journaling the linux ext2fs filesystem. In *The Fourth Annual Linux Expo* (1998).

[29] VERMA, R., MENDEZ, A. A., PARK, S., MANNARSWAMY, S. S., KELLY, T. P., AND III, C. B. M. Failure-atomic updates of application data in a linux file system. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Santa Clara, CA, 2015), FAST '15, USENIX Association, pp. 203–211.

[30] WALDSPURGER, C., AND ROSENBLUM, M. Io virtualization. *Communications of the ACM 55*, 1 (2012), 66–73.

[31] WEIL, S. bluestore, a new storage backend for ceph, one year in. In *VAULT Linux Storage and Filesystems Conference* (2017), Linux Foundation.

[32] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 307–320.

[33] WRIGHT, C. P., SPILLANE, R., SIVATHANU, G., AND ZADOK, E. Extending acid semantics to the file system. *ACM Transactions on Storage 3*, 2 (June 2007).

[34] WU, X., XU, Y., SHAO, Z., AND JIANG, S. Lsm-trie: an lsm-tree-based ultra-large key-value store for small data. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (2015), USENIX Association, pp. 71–82.