Fast Storage for File System Metadata

Kai Ren

CMU-CS-17-121
2017/09/26

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Garth A. Gibson, Chair
Dave G. Andersen
Greg R. Ganger
Brent B Welch (Google)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This research was sponsored by the Moore Foundation under grant number 2160, Los Alamos National Security, LLC under grant numbers 161465-1 and 288103, and the ISTC-CC. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: File System, Metadata Management, Log-Structured Approach, Caching

Dedicated to my family.

Abstract

In an era of big data, the rapid growth of data that many companies and organizations produce and manage continues to drive efforts to improve the scalability of storage systems. The number of objects presented in storage systems continue to grow, making metadata management critical to the overall performance of file systems. Many modern parallel applications are shifting toward shorter durations and larger degrees of parallelism. Such trends continue to make storage systems to experience more diverse metadata intensive workloads.

The goal of this dissertation is to improve metadata management in both local and distributed file systems. The dissertation focuses on two aspects. One is to improve the out-of-core representation of file system metadata, by exploring the use of log-structured multi-level approaches to provide a unified and efficient representation for different types of secondary storage devices (e.g., traditional hard disk and solid state disk). The other aspect is to demonstrate that such representation also can be flexibly integrated with many namespace distribution mechanisms to scale metadata performance of distributed file systems, and provide better support for a variety of big data applications in data center environment.

Acknowledgments

First and foremost, I am greatly indebted to my advisor, Garth Gibson for his guidance and support of my PhD research. I would also like to thank my thesis committee members Dave Andersen, Greg Ganger, and Brent Welch for their insightful feedback and comments. I would also like to express my gratitude towards my collaborators, collegues, and friends with whom I spent amazing six years: Yoshihisa Abe, Magdalena Balazinska, Vishnu Naresh Boddeti, Zhuo Chen, Christos Faloutsos, Bin Fan, Bin Fu, Gary Grider, Fan Guo, Bill Howe, Wenlu Hu, Junchen Jiang, Jin Kyu Kim, Yunchuan Kong, YongChul Kwon, Ni Lao, Lei Li, Boyan Li, Hyeontaek Lim, Liu Liu, Xi Liu, Yanjin Long, Julio Lopez, Iulian Moraru, Swapnil Patil, Andew Pavlo, Richard Peng, Amar Phanishayee, Milo Polte, Long Qin, Raja R. Sambasivan, Ilari Shafer, Julian Shun, Jiri Simsa, Pingzhong Tang, Wittawat Tantisiriroj, Yuandong Tian, Yuandong Tian, Vijay Vasudevan, Jingliang Wei, Guang Xiang, Lin Xiao, Lianghong Xu, Zi Yang, Junming Yin, Yin Zhang, Xin Zhang, Le Zhao, Qing Zheng, Dong Zhou, Zongwei Zhou. Finally, this thesis would not be possible without the support from my parents Yanjun Ren and Weinian Zhang, as well as my wife, Jieqiong Liu.

Contents

Contents ix

List of Figures xiii

List of Tables xix

1 Introduction 1

1.1 Thesis Statement . . . 2

1.2 Results Overview . . . 3

1.2.1 Out-of-core Metadata Representation . . . 3

1.2.2 In-memory Index Optimization . . . 4

1.2.3 Distributed Metadata Management . . . 4

1.3 Thesis Contribution . . . 5

2 Background and Related Work 7

2.1 Overview of File System Architecture . . . 7

2.2 In-Memory and External Memory Index . . . 8

2.2.1 In-Memory Indexes . . . 9

2.2.2 External Memory Indexes . . . 10

2.3 Metadata Management for Local File Systems . . . 11

2.3.1 Optimizations for Traditional Disk Model . . . 11

2.3.2 Optimizations for Solid State Disks . . . 13

2.4 Metadata Management for Distributed Systems . . . 13
2.4.1 Namespace Distribution . . . 13
2.4.2 Metadata Caching . . . 14
2.4.3 Large Directories Support . . . 14
2.4.4 Bulk Loading Optimization . . . 15
2.5 Storage Systems without File System APIs . . . 15

3 Metadata Workload Analysis 19
3.1 Data Sets Overview . . . 20
3.2 File System Namespace Statistics . . . 21
3.3 Dynamic Behaviors in Metadata Workload . . . 23
3.4 Statistical Properties of Metadata Operations . . . 26
3.5 Lessons from Workload Analysis . . . 29

4 TableFS: A Stacked File System Design Layering on Key-Value Store 31
4.1 Background . . . 32
4.1.1 Analysis of File System Metadata Operations . . . 32
4.1.2 LSM-Tree and Its Implementation LevelDB . . . 33
4.2 Design Overview of TableFS . . . 36
4.2.1 Local File System as Object Store . . . 36
4.2.2 Table Schema . . . 37
4.2.3 Hard Links . . . 38
4.2.4 Scan Operation Optimization . . . 39
4.2.5 Inode Number Allocation . . . 39
4.2.6 Concurrency Control . . . 39
4.2.7 Journaling . . . 40
4.2.8 Column-Style Table for Faster Insertion . . . 40
4.2.9 TableFS in the Kernel . . . 41

4.3 Evaluation . . . 43
4.3.1 Evaluation System . . . 43
4.3.2 Data-Intensive Macrobenchmark . . . 43
4.3.3 TableFS-FUSE Overhead Analysis . . . 46
4.3.4 Metadata-Intensive Microbenchmark . . . 49
4.3.5 Column-Style Metadata Storage Schema . . . 55
4.4 Summary . . . 57

5 SlimFS: Space Efficient Indexing and Balanced Read-Write Performance 59
5.1 The Analysis of Log-Structured Designs . . . 61
5.1.1 I/O Cost Analysis of Log-Structured Merge Tree . . . 61
5.1.2 Stepped-Merge Algorithm: Reducing Write Amplification in Compaction . . . 62
5.1.3 Optimizing In-memory Indexes and Filters . . . 63
5.2 The Design Overview of SlimFS . . . 65
5.2.1 The SlimFS Architecture . . . 65
5.2.2 SlimDB's Compact Index and Multi-Store Design . . . 66
5.3 Design of Compact Index and Filter in SlimDB . . . 67
5.3.1 Three-Level Index: Compact Block Index for SSTable . . . 67
5.3.2 Multi-Level Cuckoo Filter: Improve Tail Latency . . . 72
5.3.3 Implementation of SlimDB . . . 77
5.4 Analytic Model for Selecting Indexes and Filters . . . 77
5.5 Evaulation . . . 80
5.5.1 Evaluation System . . . 80
5.5.2 Full System Benchmark . . . 81
5.5.3 Compact SSTable Index Microbechmark . . . 85
5.5.4 Multi-Level Cuckoo Filters Microbenchmark . . . 87
5.6 Summary . . . 89

6 IndexFS: Metadata Management For Distributed File Systems 91
6.1 IndexFS System Design . . . 92
6.1.1 Dynamic Namespace Partitioning . . . 93
6.1.2 Stateless Directory Caching . . . 95
6.1.3 Integration with Log-Structured Metadata Storage . . . 97
6.1.4 Metadata Bulk Insertion . . . 99
6.1.5 Rename Operation . . . 100
6.1.6 Fault Tolerance . . . 102
6.2 Comparison of System Designs . . . 103
6.2.1 Table partitioned namespace (Giraffa) . . . 104
6.2.2 Replicated directories with sharded files (ShardFS) . . . 106
6.2.3 Comparison Summary . . . 109
6.3 Experimental Evaluation . . . 112
6.3.1 Large Directory Scaling . . . 114
6.3.2 Metadata Client Caching . . . 116
6.3.3 Load Balancing . . . 119
6.3.4 Bulk Insertion and Factor Analysis . . . 123
6.3.5 Portability to Multiple File Systems . . . 124
6.4 Summary of IndexFS Benefits . . . 126

7 Conclusion and Future Work 129
7.1 Future Work . . . 130

Bibliography 131

List of Figures

2.1 Distributed file system architecture. 8

3.1 The distribution of 50th, 90th, and 100th percentile, and maximum directory sizes and file sizes of collected file system samples. 21

3.2 This scatter plot demonstrates the relation between the size of each file system sample and its maximum directory size. The black line is the trend lines that predicts the relation using linear regression. . . . 22

3.3 The distribution of directory depth of collected file systems traces. Many file systems have 50% of their directories with a depth lower than 9. 23

3.4 This scatter plot demonstrates the relationship between the size of each file system sample and the average depth of objects inside the system. The average depth of the tree increases sub-linearly as the file system size grows. 24

3.5 Distribution of file system operations in LinkedIn, Yahoo! and OpenCloud traces. Most of these workloads are read-intensive, but complicated operations like `rename` also have a significant presence. 25

3.6 The distribution of each type of metadata operation in a one-day LinkedIn trace over time. The first (left) y-axis shows the fraction of each type of operation. The second (right) y-axis shows the total number of operations every minute in thousands. The variance of the distribution in the Altiscale cluster is higher than in the LinkedIn one. 26

3.7 The distribution of each type of metadata operations in a 125-day Altiscale trace over time. The variance of the distribution in the Altiscale cluster is higher than in the LinkedIn one. 27

3.8 Distribution of metadata operations in the LinkedIn trace by length of accessed pathname. 27

3.9 Distribution of access frequency of pathnames. 28

4.1 File and directory metadata structures 33

4.2 LevelDB represents data on disk in multiple SSTables that store sorted key-value pairs. Each solid rectangle represents an SSTable. LevelDB uses a multi-level structure to grow an LSM-tree store (with a growth factor $r = 10$). 34

4.3 (a) The architecture of TableFS. A FUSE kernel module redirects file system calls from a benchmark process to TableFS, and TableFS stores objects into either LevelDB or a large file store. (b) When we benchmark a local file system, there is no FUSE overhead to be paid. 36

4.4 An example illustrates table schema used by TableFS's metadata store. The file with inode number 4 has two hard links, one called "apple" from directory *foo* and the other called "pear" from directory *bar*. . . 38

4.5 Column-style stores index and log tables separately. Index tables contain frequently accessed attributes for file lookups and a pointer to the location of full file metadata in the most recent log file. Index tables are normally compacted while log tables are rarely or never compacted, reducing the total work for TableFS. 40

4.6 Three different implementations of TableFS: (a) the kernel-native TableFS, (b) the FUSE version of TableFS, and (c) the library version of TableFS. In the following evaluation section, (b) and (c) are presented to bracket the performance of (a), which was not implemented. 42

4.7 The normalized elapsed time for kernel building. All elapsed time is divided by the minimum value (1.0 bar). The legends above each bar show the minimum value in seconds. 45

4.8 The elapsed time for both the entire run of Postmark and the transactions phase of Postmark for the four tested file systems. 45

4.9 Average throughput of each type of operation in the Postmark benchmark. 46

4.10 The elapsed time for creating 1M zero-length files on three versions of TableFS. 47

4.11 Total disk traffic associated with Figure 4.10 47

4.12 Changes of total size of SSTables in each level over time during the creation of 1M zero-length files for three TableFS models. TableFS-Sleep illustrates similar compaction behavior similar to TableFS-FUSE. . . 48

4.13 Average throughput during four different workloads for five tested systems. All tests were run for three times, and the coefficient of variation was less than 1%. 50

4.14 Total number of disk read/write requests during 50%Read+50%Write query workload for five tested systems. 51

4.15 Total run-time of three *readdir* workloads for five tested file systems. 52

4.16 Throughput of all four tested file systems while creating 100 million zero-length files. TableFS-FUSE is almost 10× faster than the other tested file systems in the later stage of this experiment. The data is sampled in every 10 seconds and smoothed over 100 seconds. The vertical axis is shown on a log scale. 53

4.17 Average throughput in the create and query workloads on an Intel 520 SSD for five tested file systems. 54

4.18 Total number of disk requests and disk bytes moved in the query workload on an Intel 520 SSD for five tested file systems. 55

4.19 Average benchmark bandwidth when inserting 3 million entries with different sizes into the column-style storage schema and the LevelDB-only on a single server. 56

5.1 Illustration of the compaction of LSM-tree. 61

5.2 Illustration of Stepped-Merge algorithm. 63

5.3 Illustration of the basic index format of LevelDB's SSTable and its read path. The keys follow a semi-sorted order, so each key has two parts: the prefix (red) and the suffix (black). 64

5.4 The use of multi-store design in SlimDB. Filters and indexes are generally in-memory, and for a large store SSTables are mostly on disk. 66

5.5 ECT transforms a list of sorted key hashes into a radix tree that only keeps the shortest prefix of each key that is enough to distinguish it from other keys. 68

5.6 An example three-level block index for a SlimDB SSTable. 69

5.7 Illustration of Cuckoo Hashing. 73

5.8 Illustration of integrating cuckoo filters with multi-level indexes and a secondary table. If a key has a hash collision with some key in the primary hashing table, the key will be put into the secondary hashing table. Each lookup first checks in the secondary table and then the primary table. 74

5.9 The per-operation cost estimated by the model. 79

5.10 *The average file creation throughput and write amplification of tested system over the entire create phase.* 82

5.11 This figures shows the instantaneous file creation throughput during the create phase. Each data point shows the average throughput within 100 seconds time window, and is sampled when every 1 million new files are created. 83

5.12 This figures shows the instantaneous file `stat` throughput during the query phase. Each data point shows the average throughput within a 10-second time window, and samples are taken when every 10,000 file `stat` requests are issued. 84

5.13 The latency distribution of `stat` operations for all tested systems during the query phase. 85

5.14 Average lookup throughput of three filters under different duplication ratios. 89

6.1 The IndexFS metadata system is middleware layered on top of an existing cluster file system deployment (such as PVFS or Lustre) to improve metadata and small file operation efficiency. It reuses the data path of the underlying file system and packs directory entries, file attributes and small file data into large immutable files (SSTables) that are stored in the underlying file system. 92

6.2 This figure shows how IndexFS distributes a file system directory tree evenly into four metadata servers. Path traversal makes some directories (e.g. the root directory) more frequently accessed than others. Thus stateless directory caching is used to mitigate these hot spots. . 94

6.3 Orphaned loop from two `rename` operations. 102

6.4 Giraffa stores its metadata in HBase, which partitions its table as a B-Tree. Each file or directory is mapped to an unique row at one HBase region server. The current implementation of Giraffa does not have hierarchical permission checking so no pathname resolution is performed.
. 105

6.5 ShardFS replicates directory lookup state to all metadata servers so every server can perform path resolution locally. File metadata and non-replicated directory metadata is stored at exactly one server determined by a hash function on the full pathname.
. 106

6.6 IndexFS on 128 servers deliver a peak throughput of roughly 842,000 file creates per second. The prototype RPC package (Thrift [thr]) limits its linear scalability. 114

6.7 IndexFS achieves steady throughput after distributing one directory hash range to each available server. After scale-out, throughput variation is caused by the compaction process in LevelDB. Peak throughput degrades over time because the total size of the metadata table is growing, so negative lookups do more disk accesses. 115

6.8 Average aggregate throughput of replaying 1 million operations per metadata server on different number of nodes using a one-day trace from a LinkedIn HDFS cluster. 116

6.9 Latency distribution of update operations (a) and lookup operations (b) under different caching policies ($6.4e2$ means 6.4×10^2). Rate-based policies offer the best average and 99% latency, which yields higher aggregate throughput. 118

6.10 Performance comparison among IndexFS, ShardFS, and Giraffa creating and stating zero-byte files with 64 server machines and 64 client machines. 120

6.11 Contribution of optimizations to bulk insertion performance on top of PVFS. Optimizations are cumulative. 123

6.12 Per-server and aggregated throughput during mdtest with IndexFS layered on top of Lustre (on Smog), HDFS (on Kodiak), and PanFS (on Susitna) on a log scale. HDFS and Lustre have only one metadata server. 125

6.13 The aggregate write throughput for the N-N checkpointing workload. Each machine generates 640 GB of data. 126

List of Tables

3.1 This data summarizes the specifics of the data collected from 96 file systems. 20

3.2 Percentage of rename operations that generate conflicts in each collected trace. 29

4.1 *Average throughput when reading 5 million 320B entries from the column-style schema and original LevelDB-only on a single server.* 57

5.1 Key-value store schema used in SlimFS. 65

5.2 The space and disk access cost of using three types of indexes. CF means cuckoo filter. TL means three-level SSTable index. MLCF means multi-level cuckoo filter. 78

5.3 Hardware configuration for experiments. 81

5.4 The memory consumption of different SSTable indexes measured as bits per key, for various patterns of prefix groups (3 fixed sized and a Zipfian distribution 87

5.5 Average SSTable index lookup speed for SlimDB and the original LevelDB in thousands of lookups per second. 87

5.6 RAM usage and false positive rate of different filters. 88

5.7 Bulk insertion throughput of three filters under different duplication ratio in millions of inerstions per second. 89

6.1 The schema of keys and values used by IndexFS. Only the value of a directory contains the "mapping" data, which is used to locate the directory partition's server. 98

6.2 Summary of design choices made by three metadata services. 110

6.3 Three clusters used for experiments. 113

Chapter 1

Introduction

The last decade has seen tremendous innovation and changes in computing systems, as rapid growth of data produced by many commercial companies and scientific organizations drive efforts to scale out computer clusters. In such an era of big data, new trends in technology and application workloads call for scaling the metadata management in the modern distributed file systems.

One of the obvious trend is that new data sources and instruments (e.g. the web, gene sequencers, and wireless sensors) will continue to produce rapidly increasing amounts of information. Recent studies have shown that the size of local file systems is soon expected to achieve and exceed billions of objects [Whe10]; many cloud storage systems already hold more than trillions of objects [s3t, azu]. The steadily increasing number of objects stored in storage systems continually stresses the system and demands higher scalability.

On the other hand, massive parallelism has become more prevalent in today's applications since large collections of cluster resources are more approachable to everyone thanks to cloud computing providers like Amazon, Microsoft and Google. Workloads generated by these parallel applications are very diverse, ranging from batched scientific computing to interactive query processing. Despite a plethora of scalable storage systems such as key value stores and distributed databases, distributed file systems continue to be the dominant interface for many parallel applications to manage data in clusters. Many of them require concurrent and high-performance metadata operations, which support the need for a scalable metadata service. For example, file-per-process check-pointing in many HPC applications requires the metadata service to absorb a huge number of concurrent file creations within a short period [BGG+09]. Another example, storage management, produces a read-intensive meta-

data workload that typically scans the metadata of the entire file system to perform administrative tasks [Jea11, Lea09]. Previous studies [RKBH13, CAK12, WN13] have also shown that many large-scale data analytic applications process many small and transient objects that are metadata intensive workloads. In particular, frameworks that support these applications (e.g., Impala [Mar12], Spark [ZCD+12], and Sparrow [OWZS13]) are shifting toward shorter task durations and larger degrees of parallelism to provide low latency. These applications will generate more metadata intensive workload to the underlying storage systems.

Modern distributed storage systems commonly use an architecture that decouples metadata access from file data accesses [GGL03, HDF, WUA+08]. File system metadata is structured by hierarchical semantics and often accessed in small units, which makes it difficult for the file system to preserve data accessing locality both machine-wise and disk-wise. Popular distributed file systems such as HDFS [HDF] and the first generation of Google file system [GGL03] have used centralized single-node metadata services and focused on scaling only the data path. However, single-node metadata server design limits the scalability of the file system in terms of the number of stored objects and concurrent accesses [Shv10]. Federating independent metadata services employs multiple server nodes but does not ensure load balancing among them. Moreover, all too often the data-to-metadata ratio is not high, since even in large file system installations, most files are small [Day08, WN13, HBD+14]. Previous research [WBML06, Fik, Gir13] has proposed several system designs to scale out file system metadata service. Some of these systems use optimizations tailored to particular workloads or relaxed metadata operations semantics in order to enhance scalability. By examining several real-world file system metadata traces, we find that metadata workloads are more diverse and some assumptions made in previous works do not always hold. The main motivation of this dissertation is therefore to scale distributed file systems to meet the demand of the diverse large-scale metadata intensive workloads.

1.1 Thesis Statement

The use of multi-level log-structured approach can provide a unified and efficient out-of-core representation for file system namespace metadata on modern storage devices including hard disks and solid state disks. Such representation can also be flexibly integrated with namespace distribution mechanisms to scale out namespace metadata performance for distributed file systems.

1.2 Results Overview

This dissertation focuses on scaling file system metadata management from two aspects: one is to scale up the performance of the local metadata store, which essentially improves the on-disk representation of file system metadata and the use of compact in-memory index; the other is to scale out metadata management in a distributed environment by carefully partitioning file system namespace and using distributed leasing techniques to achieve load balancing.

1.2.1 Out-of-core Metadata Representation

While local file systems have been studied extensively, recent advances in indexing data structures brings new opportunities. Techniques like log-structured merge (LSM) trees and compact in-memory indexes are widely used in modern key-value stores to serve small random requests. It is likely that these data structures are more suitable for file system metadata workloads because their aggressive aggregations of metadata updates fully utilize disk bandwidth without sacrificing lookup performance.

We introduce a stackable file system architecture called TableFS that represents file system metadata into sparse on-disk key-value tables. Its modular design can leverage existing key-value stores to flexibly plug in different key value stores to meet requirements of various metadata workloads. By carefully choosing data schema, TableFS packs file and directory attributes closer on the disk surface to enhance data locality for scan and point lookup performance. TableFS also supports a second table schema for write heavy workloads, which uses another indirection to avoid unnecessary background compactions in the underlying key-value store. Our library interface is thin enough such that little overhead is added. By stacking, TableFS asks only for efficient large file allocation and access from the underlying local file system. By using an LSM tree, TableFS ensures metadata is written to disk in large, non-overwrite, sorted and indexed logs. Our implementation, even hampered by FUSE overhead, library code overhead, compaction overhead, and pessimistically padded inode attributes, can outperform popular Linux file systems including Ext4, XFS and Btrfs by 50% to as much as 1000% for metadata-intensive workloads.

1.2.2 In-memory Index Optimization

Through the evaluation on TableFS, we identify several places where the LSM-tree can be further improved: (1) reducing write amplifications caused by compactions, (2) reducing read amplifications on solid-state disks, and (3) reducing the size of in-memory index.

To address these issues, the focus is to reduce the size of the in-memory index and combine it with another write-optimized on-disk layout to achieve better balance between read and write performance. The core idea reducing the in-memory index size is to use a hash-based key schema. By using compression, it only costs less than 2 bits per key to index the location of any key in an SSTable. A data structure called a multi-level cuckoo filter is proposed to improve the worst-case latency and speed up the in-memory searching procedure. By taking advantage of file system semantics, maintaining the multi-level cuckoo filter on-disk incurs no additional overhead than traditional Bloom filters used in LevelDB. By combining two novel index techniques with a write-optimized key-value layout called Stepped-Merge algorithm, the experiments show that SlimFS can improve write and read throughput of original TableFS by up to three times and up to two times, respectively.

1.2.3 Distributed Metadata Management

The third study is to design a distributed metadata service that scales over hundreds of machines and delivers millions of metadata operations per second. We have demonstrated a middleware solution called IndexFS, which can be easily layered on top of existing file systems including PanFS, Lustre and HDFS to help improve performance of their original non-scalable metadata path.

The main challenge of designing such a system is to overcome performance bottlenecks to scale the entire system, thereby gaining sustainable high throughput and low latency for metadata operations. A variety of techniques have been developed to improve IndexFS's performance: 1) IndexFS incrementally partitions the file system namespace on a per-directory basis, preserving disk and server locality for small directories for improving the efficiency of scan operations as well as load balancing for large directories; 2) We also proposed two client-based caching techniques: one allows stateless consistent metadata caching to avoid hot spots on the server side, and the other uses bulk insertion of newly created namespace for ingestion heavy workloads such as N-N Check-pointing; 3) We also utilized an optimized log-structured on-disk data layout to store metadata and small files efficiently.

To demonstrate the feasibility of our approach, we implemented a prototype middleware service called IndexFS that incorporates the above namespace distribution and caching mechanisms as well as the on-disk metadata representation. Existing cluster file systems, such as PVFS, HDFS, Lustre, and PanFS, can benefit from IndexFS without requiring any modifications to the original system. we evaluated the prototype on multiple clusters consisting of up to 128 machines. Our results show promising scalability and performance: IndexFS, layered on top of PVFS, HDFS, Lustre, and PanFS, can scale almost linearly to 128 metadata servers, performs 3000 to 10,000 operations per second per machine, and outperforms the underlying file system by 50% up to two orders of magnitude as the number of servers scales in various metadata intensive workloads.

1.3 Thesis Contribution

The thesis makes the following contributions:

- TableFS is a stackable local file system architecture where the underlying key-value storage and object storage are pluggable to meet the requirements of different file system workloads. Under this architecture, we demonstrate that packing file and directory attributes into log-structured key value store can effectively enhance data locality and outperform modern Linux file systems.
- SlimFS further pushes the limit of using in-memory index to improve read-write performance balance for file system metadata workloads. Through algorithmic engineering and utilizing file system semantics, the compact three-level index for SSTable only costs 2 bits per key, and multi-level cuckoo hashing can help bound the number of disk reads in the worst case. The two indexing data structures can be easily integrated with a write-optimized data layout such that SlimFS achieves better performance than original TableFS for both read and write operations.
- IndexFS demonstrates an efficient combination of our scale out indexing technique with a scale-up metadata representation to enhance the scalability and performance of metadata service. By incorporating client caching with minimal server state, it is able to achieve load balancing and high file creation throughput. Its portable design also works with many existing file system deployments with a few configuration changes to the file system or the systems software on compute nodes.

In the following sections, we first present two local file system designs TableFS and SlimFS, and then discuss the design of distributed metadata service IndexFS. However, the chronological order of the three works is actually TableFS, IndexFS, and SlimFS. Thus, many performance optimization techniques proposed in SlimFS were not evaluated in any experiment in the section discussing IndexFS.

Chapter 2

Background and Related Work

Before the discussion of my thesis works, this chapter first presents a background on the file system architecture used in this dissertation. This chapter also discusses how previous literature has motivated the development of our metadata management systems. Given the vast body of related works in storage systems, this discussion focuses on techniques most relevant to file system metadata management, and compares them with the TableFS, SlimFS and IndexFS design.

Section 2.1 presents an overview of the architecture of modern distributed file systems used by IndexFS. Section 2.2 summarizes the design trade-offs of a variety of core data structures used as building blocks in many storage systems. Section 2.3 and Section 2.4 summarize metadata management in local and distributed file systems respectively. Section 2.5 discusses why directly reusing database systems such as relational databases and key-value data stores

2.1 Overview of File System Architecture

The architecture used by IndexFS inherits major properties of modern distributed file systems such as NASD [GNA+98], GFS [GGL03] and HDFS [HDF]. As shown in Figure 2.1, it is a client-server architecture where clients are gateways for data accessing and the servers are responsible for persistent data storage. This architecture also decouples metadata management from data management by having separate metadata servers and data servers. Metadata servers manages the file system namespace and metadata associated with files and directories including data location, time stamps, permissions and other attributes. When reading or writing a file, clients first com-

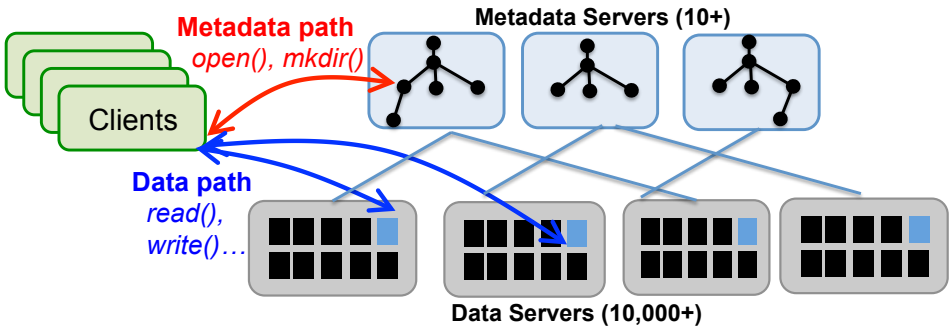


Figure 2.1: Distributed file system architecture.

municate with metadata servers to locate file data and then subsequently interact directly with data servers. This allows efficient parallel data transfers between a large number of clients and data servers without frequent communications with metadata servers. To simplify the implementation and design, traditional file systems mostly have only a centralized single-node metadata server with backup servers in case of failure, or statically partition the namespace such that a workload can easily apply all work to one server. IndexFS instead focuses on scaling out metadata management by distributing metadata across many servers. While data block management is also one of metadata servers' main functions in many file systems, this dissertation emphasizes the design of namespace metadata management.

2.2 In-Memory and External Memory Index

Index data structures, as key components to the storage system, have direct impact on its interface and performance. Many storage systems equip both an in-memory index and an external memory index for different purposes. For example, in-memory indexes are often used to serve requests to popular data items by taking advantage of faster memory speed; external memory indexes are used for maintaining orderly and persistent data on the secondary storage.

The discussion of previous related work on in-memory and external indexes concentrates on the trade-offs on the following aspects:

- *Interfaces*: Many indexes used by storage systems are single-dimension, where the items are indexed by one primary key. Depending on the usage, items of the

index may contain a value field or not (key-only indexes such as bitmaps can be used for membership test). Keys can also be stored by different orders in the index such as sorted order or hashing order, which may affect the performance of different operations. Common operations include `PUT`, `GET`, `DEL`, and `SCAN`. Many indexes can only efficiently support a subset of these operations.

- *Resource Usage*: Another important aspect about the indexes is their cost on different machine resources include CPU, memory and disk I/O and capacity. How to balance the usage over different resources affects the choice of indexes for the storage systems.

2.2.1 In-Memory Indexes

In-memory indexes are often used along with external index structure to reduce unnecessary disk accesses for read operations. To reduce memory cost, these indexes are often designed with compact representations. A Bloom filter [Blo70] is an index structure used to represent a set of keys for set-membership tests. Bloom filter can be used to reduce disk accesses of negative lookups, and locate the on-disk position of a key-value pair. By allowing false positives, its size is very compact. Typically, it requires 8 or more bits per key to bound the false positive rate within 1%. There are variants like Bloomier filters can generalize Bloom filters to represent functions. But these structures are more complex and require more space than Bloom filter or cuckoo filter [CKRT04, FAKM14]. Perfect hashing indexes such as ECT [LAK13] used in SILT [LFAK11], and CHD [BBD09] use fewer than 2.5 bits per key to map any given key in a set to a distinct location. Such indexes support positive lookups but do not answer negative lookups, and rely on batching to perform updates efficiently. Set separator [FZL+13] is another space-efficient index structure that maps a large set of keys into a small range of values. Its per-key memory overhead is only proportional to the size of its value range.

There are also many memory indexes combined with caching algorithms. MICA [LHAK14] boosts performance on multi-core CPUs by sharding data to dedicated cores to avoid synchronization and using simplified FIFO caching algorithm to prevent contentions among cores. MemC3 [FAK13] combines optimistic cuckoo hashing and CLOCK cache eviction algorithms for read-mostly workloads. Besides these hashing-based key-value stores, Masstree [MKM12] applies extensive optimizations for cache locality and optimistic concurrency control, but uses very different techniques to support range queries because it is a B+-tree variant.

2.2.2 External Memory Indexes

The B-Tree [BM72] is a fundamental external index structures and is used in nearly all storage systems. Theoretically, external indexes are analyzed in the standard I/O model [AV88]. In the model, N denotes the size of the data set, M is the internal memory size, and B represents the disk block size. All parameters are measured in terms of data records (or key value pairs). This model mainly measures the number of disk blocks accessed (I/Os). By grouping sorted data records in disk blocks, the B-Tree occupies $O(\frac{N}{B})$ blocks. It takes $O(\log_B N)$ I/Os for point queries and $O(\log_B N + \frac{K}{B})$ I/Os for range reporting queries. In practice, storing top levels of the B-Tree in the memory, it usually requires only one or two I/Os to access the desired data record.

Modern magnetic disks have increased in capacity and block size constantly over the years by increasing storage density, but still provides only a limited number of random I/Os per second. An interesting observation made in the log-structured file system paper (LFS) [RO91] is that buffering a number of updates in memory and performing the updates in batches can significantly lower the amortized cost. The Buffer-tree [Arg95] was the first method along this research line to use a large in-memory buffer associated with internal nodes of the B-tree to perform update/insert operations in a "lazy" manner. The Log-Structured Merge tree (LSM tree) [OCGO96] uses an in-memory buffer differently by applying logarithmic method [BS80] to the B-Tree. LSM trees represents a large B-tree in a collection of B-trees of size up to M, sM, s^2M, ..., s^LM and only maintains the smallest B-tree in memory as buffer. It has a background operation called "compaction", which periodically merges these on-disk B-trees as needed to reduce the number of disk reads for future lookups or to reclaim disk space. Standard analysis shows that the amortized insertion cost of an LSM-tree is $O(\frac{l}{B} log_l \frac{N}{M})$. By using fractional cascading [CG86], B^ϵ Tree is a B-Tree augmented with per-node buffers. New items are inserted in the buffer of the root node of a B^ϵ-tree. When a node's buffer becomes full, messages are moved from that node's buffer to one of its children's buffers. The leaves of the B^ϵ-tree store key-value pairs, as in a B-tree. Point and range queries behave similarly to a B-tree, except that each buffer on the path from root to leaf must also be checked for items that affect the query. This further improves the cost of range reporting queries to be $O(log_l \frac{N}{M} + \frac{K}{B})$ without affecting the (asymptotic) size of index and update cost.

In practice, many modern implementations of LSM-trees improve the "compaction" operation for throughput and latency. In LevelDB [Lev11], each level of a B-tree is represented as a set of static sorted tables with disjoint key ranges. Each compaction

operation only merges a limited number of sorted tables, which lowers the impact of compaction on other concurrent operations. If bounding the variance on insert response time is critical, compaction algorithms can be more carefully scheduled, as is done in bLSM[SR12]. HyperLevelDB [Hyp13b] greedily chooses a set of sorted tables with the smallest merging cost during each compaction, and the reduction of write amplification depends on the workload. VT-trees [SSM+13] exploits the sequentiality in the workload by avoiding always copying old SSTable content into new SSTables during compaction. These trees add another layer of pointers so new SSTables can point to regions of old SSTables, reducing data copying but requiring extra seeks and eventual defragmentation.

2.3 Metadata Management for Local File Systems

This section summarizes previous works on performance improvement of local file systems. This section first talks about techniques used to optimize file system performance based on a traditional disk model that assumes symmetric performance for disk reads and writes. With increasing popularity of solid state disks, there are also many recent works that explore the intrinsic properties of solid state disks that differ from traditional magnetic disks: asymmetric read and write performance, and wearing properties.

2.3.1 Optimizations for Traditional Disk Model

Many early file systems (e.g. UNIX file system, fast file system, Linux Ext file systems [MCB07]) stored directory entries in a linear array in a file and inodes in simple on-disk tables at a fixed location, separated from the data of each file. Inode bitmaps are also used to describe free inodes. The drawback of this design is that the number of inodes is fixed, and updating an inode requires updating multiple locations on the disk. Clustering within a file was pursued aggressively, but for different files clustering was at the granularity of the same cylinder group. It has long been recognized that small files can be packed into the block pointer space in inodes [MT84]. C-FFS [GK97] takes packing further and clusters small files, inodes and their parent directory's entries in the same disk readahead unit — the track. A variation on clustering for efficient prefetching replicates of inode fields in directory entries, as is done in NTFS [Cus94].

Later file systems such as XFS [Swe96] and Ext4 [MCB07] aggressively and pervasively use B+ trees to scale many file structures: free space maps, file extent maps, directory entry indexes and dynamically allocated inodes. To reduce the on-disk size of these structures, XFS also partitions the disk into allocation groups, clusters allocation in an allocation group, and uses allocation group relative pointers with fewer bytes.

Beginning with the Log-Structured File System (LFS) [RO91], file systems have exploited write allocation methods that are non-overwrite, log-based and deferred. Variations of log structuring have been implemented in NetApp's WAFL, Sun's ZFS and BSD UNIX [ZFS, HLM94, SBMS93]. This dissertation mainly focuses on the disk access performance implications of non-overwrite and log-based writing, although the potential of strictly ordered logging to simplify failure recovery in LFS has been emphasized and compared to various write ordering schemes such as Soft Updates and Xsyncfs [MG99, NVCF08, SGM+00].

Btrfs [Kar09, RBM12] is the newest Linux file system. Inspired by Rodeh's copy-on-write B-tree [Rod08], Btrfs copies any B-tree node to an unallocated location when it is modified. Provided the modified nodes can be allocated contiguously, B-tree update writing can be highly sequential; however, more data must be written than is minimally needed (write amplification), because modifying a B-tree node may cause rewriting its ancestry nodes inside the B-tree.

Partitioning the contents of a file system into two groups — a set of large file objects and all of the metadata and small files — has been explored in hFS [ZG07]. In this design, large file objects do not float as they are modified, and a modified log-structured file system approach and an in-place B-Tree is used to to manage metadata, directory entries and small files. TableFS has this split as well, with large file objects handled directly by the backing object store. In LevelDB's partitioned LSM tree, metadata updates are approximately log structured. However, TableFS uses a layered approach and does not handle disk allocation, showing that metadata performance of widely available and trusted file systems can be greatly improved even in a less efficient stacked approach.

Similar to TableFS, there are also a few recent works that build file systems on top of key-value stores (indexed data structures). BetrFS [WJea15] stores both file system metadata and data blocks into an in-kernel version of a B^ϵ tree. As explained in the previous section, B^ϵ trees utilize additional on-disk indexes (fractal cascading indexes), from which it gets better asymptotic read performance than from LSM trees. BetrFS also uses full pathnames as the primary key for all metadata, which brings faster point queries, but makes it difficult to implement an efficient rename

operation. A naive implementation of a rename operation needs to copy all files under a subtree.

2.3.2 Optimizations for Solid State Disks

Many modern Linux file systems such as Ext4, XFS, and Btrfs, have been tuned for flash storage by explicitly managing wear leveling of flash devices. Researchers have also proposed several new general-purpose file systems for flash storage. DFS [JBFL10] is a file system that directly manages flash memory by leveraging functions (e.g., block allocation, atomic update) provided by FusionIO's ioDrive. Nameless Write [ZAADAD12] also removes the space allocation function in the file system and leverages the FTL space management for space allocation. OFSS [LSZ13] proposes to directly manage flash memory using an object-based FTL, in which object indexing, free space management and data layout can be optimized with flash memory characteristics. F2FS [LSHC15] is a log-structured file system which is designed for flash storage. It optimizes data layout in flash memory, e.g., hot/cold data groupings.

The file systems mentioned above concentrate on improving data accesses on flash storage. There are other works that instead improve metadata accesses, which are frequent scattered small write patterns. ReconFS [LSW14] uses an inverted index and asynchronous snapshotting of the namespace to reduce namespace metadata writeback size while providing hierarchical namespace access. ReconFS ensures consistency by embedding an inverted index in each page, eliminating the writes of the pointers (indexing for directory trees). To guarantee persistence, it asynchronously compacts and logs scattered small updates to the metadata persistence log to reduce write size. The inverted indexes and logs are used respectively to reconstruct the structure and the content of the directory tree in case of system failures. KVFS [SSM+13] is built on a transactional variation of an LSM tree, called a VT-tree. VT-trees exploit sequentiality in the workload by adding another indirection (called stitching) to avoid merge sorting all aged SSTables during compaction. This design trades more disk reads for fewer disk writes, which especially benefits solid state disks.

2.4 Metadata Management for Distributed Systems

2.4.1 Namespace Distribution

PanFS [WUA+08] uses a coarse-grained namespace distribution by assigning a sub-tree (called a volume) to each metadata server (called a director blade). PVFS [RL06] is more fine-grained: it spreads different directories, even those in the same sub-tree, over different metadata servers. Ceph [WBML06] dynamically distributes collections of directories based on server load. The distributed directory service [DH06] in Farsite [ABC+02] uses tree-structured file identifiers for each file. It partitions the metadata based on the file identifier prefix, which simplifies the implementation of rename operations. Lustre [Lus] mostly uses one special machine for all metadata, and is developing a distributed metadata implementation. IBM GPFS [SH02] is a symmetric client-as-server file system which distributes mutation of metadata on shared network storage, provided the workload on each client does not generally share the same directories.

2.4.2 Metadata Caching

For many previous distributed file systems, including PanFS, Lustre, GPFS, and Ceph, clients employ a name space cache and an attribute cache for *lookup* and *stat* operations to speed up path traversal. Most distributed file systems use cache coherent protocols in which parallel jobs in large systems suffer cache invalidation storms, causing PanFS and Lustre to disable caching dynamically. PVFS, like IndexFS, uses a fixed-duration timeout (100 ms) on all cached entries, but PVFS metadata servers do not block mutation of leased cache entries. Lustre offers two modes of metadata caching depending on different metadata access patterns [Sch03]. One is a writeback metadata caching that allows clients to access a subtree locally via a journal on the client's disk. This mode is similar to bulk insertion as used in IndexFS, but IndexFS clients replicate the metadata in the underlying distributed file system instead of on the client's local disk enabling failover to a remote metadata server. Another mode offered by Lustre and PanFS is to execute all metadata operations on the server side without any client caching during highly concurrent accesses. Farsite [DH06] employs field-level leases and a mechanism called a disjunctive lease to reduce false sharing of metadata across clients, thus mitigating metadata hotspots. This mechanism is complementary to our approach. However, it maintains more state about the owner of the lease at the server in order to later invalidate the lease.

2.4.3 Large Directories Support

A few cluster file systems have added support for distributing large directories, but most spread out the large namespace without partitioning any directory. A beta release of OrangeFS, a commercially supported PVFS distribution, uses a simplified version of GIGA+ to distribute large directories on several metadata servers [Mea11]. Ceph uses an adaptive partitioning technique for distributing its metadata and directories on multiple metadata servers. IBM GPFS uses extensible hashing to distribute directories on different disks on a shared disk subsystem and allows any client to lock blocks of disk storage. Shared directory inserts by multiple clients are very slow in GPFS because of lock contention. As well, GPFS is only able to deliver high read-only directory read performance when directory blocks are cached on all readers [PG11].

2.4.4 Bulk Loading Optimization

Considerable work has been done to add bulk loading capability to new shared-nothing key-value databases. PNUTS [SCS+08] offers bulk insertion of range-partitioned tables. It attempts to optimize data movement between machines and reduce transfer time by adding a planning phase to gather statistics and automatically tune the system for future incoming workloads. The distributed key-value database Voldemort [SKG+12], like IndexFS, partitions bulk-loaded data into index files and data files. However, it utilizes offline MapReduce jobs to construct the indexes before bulk loading. Other databases such as HBase [Fou] use a similar approach to bulk load data.

2.5 Storage Systems without File System APIs

Previous database research has explored many techniques related to index building and external data storage layout, in order to improve performance for various database workloads. For example, a number of column-oriented database systems, including MonetDB [BGvK+06] and C-Store [SAB+05], were proposed to optimize performance on certain workloads such as read-intensive analytical processing workloads. The physical layout of column-oriented database systems (vertical partitioned tables) allows specific optimizations in query execution and data compression, bringing advantages over traditional row-oriented databases. Adaptive indexing and database

cracking [GK10, IKM07] were proposed to allow on-the-fly physical data reorganization in databases, as a collateral effect of query processing. Indexes are then built incrementally, adaptively, and on demand as part of database operators; the more queries processed, the more the relevant indexes are optimized.

Traditional "one size fits all" relational database systems (RDBMS) do not meet the scalability and performance requirements of modern data-intensive applications [Sel08, SC05]. To alleviate these concerns, new key-value data stores were designed from scratch using only those database semantics and functions that were required by the target applications. Notable examples are Google's BigTable [CDG+06], Amazon's Dynamo [DHJ+07], and SciDB [CMKL+09]. These data stores scale out, typically, by supporting only a subset of an RDBMS's transactional ACID semantics. Different stores relax different semantics and offer different properties. For example, BigTable and Dynamo limit atomicity to per-object or per-row mutations. Dynamo and SciDB relax consistency through eventual application-level inconsistency resolution or weak integrity constraints. To support stronger transaction semantics in a large scale, distributed, geographic environment, some data stores (e.g. MegaStore [BBC+11] and Spanner [CDE+12]) use optimized consensus protocols with special hardware support, and also favor read operations over write operations.

The transaction management policies used in IndexFS share some similarities with database systems. The LevelDB local storage backend used by IndexFS uses the multiversion concurrency control protocol (MVCC [BG81]) to manage concurrent data access. H-store [SMA+07] is a modern main memory databases that uses a sophisticated concurrency control strategy. H-store classifies transactions into multiple classes where single-sited or sterile transactions are executed without locks. For non-sterile and non-single-sited transactions, H-store uses an optimistic method to execute distributed transactions and dynamically escalates to a more sophisticated strategy when the percentage of aborted transactions exceeds a certain threshold.

Notably, there have been previous attempts to unify file systems and databases, or build file systems on top of databases. Olson's Inversion file system [Ols93], uses a transactional database (Postgres) to implement a file system that provides transactional guarantees, rich queries, and fine-grained versioning. However, because of large size, complexity and few optimizations for file system workloads, Inversion file system's performance is slow. Several other earlier works attempted to strengthen transactional semantics of file systems by either using a database internally [WSSZ07], or providing file system specific transactional semantics [HMSC87, Kas04, LR04, SSM+13].

Giraffa [Gir13] represents file system metadata as a large inode table and stores them into an external distributed key-value store, HBase [Fou], reusing HBase's partitioning and load-balancing functionality. The key schema used by Giraffa is the full pathname with the depth (of a particular file system object) in the namespace tree as a prefix. It sacrifices semantics (no recursive permission checking) to reduce cost for multiple pathname lookups on servers.

Similarly, CalvinFS [AT15] leverages a distributed database system for metadata management. By using the full pathname hash value as a primary key, file metadata are partitioned and replicated across a (shared-nothing) cluster of independent servers. File metadata operations are transformed into distributed transactions. CalvinFS is optimized for single-file operations, and other metadata operations such as directory renames and permission changes need to recursively modify all entries under the affected subtree.

Chapter 3

Metadata Workload Analysis

This chapter conducts a deep study into the file system metadata workload. The goal of this workload study is to determine which properties in metadata workloads that shed light on the file system design. The workload data is collected from 96 file system installations whose size ranges from personal desktops to a cluster consisting of thousands of nodes. The chapter mainly studies the metadata workload of the following three aspects, and tries to answer the questions around them that affect file system design choices.

- **Namespace Structure** (Section 3.2). Understanding the structure properties of file system namespace helps with file system on-disk layout. For example, the average directory size and file size can decide the threshold for embedding directory entries and file data inside the inode.

- **Dynamic Behavior** (Section 3.3). Dynamic behavior depicts how metadata workloads evolve over time, especially the metadata and data access patterns. This affects many file system's components including inode cache, data cache and background cleaning process.

- **Individual Metadata Operations** (Section 3.4). There are also statistical properties related to individual metadata operations that are important. For example, the conflict ratio among concurrent `rename` operations will determine what kind of protocol should be chosen for `rename`.

3.1 Data Sets Overview

Our workload traces were collected from a variety of production file systems which can be categorized into three types: traditional enterprise storage (e.g., NFS workloads), high performance computing (HPC) (e.g., MPI workloads), and data-intensive computing (DISC) (e.g., Hadoop cluster traces from many Internet companies). The gathered traces were either reproduced from previous works [WN13, FTXG11] or collected by my research group through personal communication. Table 3.1 summarizes the information of the collected workload traces. These storage systems include file servers from Carnegie Mellon Unversity Parallel Data Lab [Day08], 65 customer installations from Panasas [WN13], Hadoop clusters from Yahoo!, Facebook, LinkedIn, Cloudera, Altiscale, and CMU OpenCloud. The size of the storage systems in the traces ranges from one single machine to thousands of machines. Usually, PDL file servers have a few nodes. The rest of traces were gathered from medium to large file system installations consisting of a few dozen to thousands of machines, particularly the Hadoop clusters, which were usually comprised of at least hundreds of machines.

Sources	# FSs	File System	Log Type	Duration
PDL servers	6	WAFL	Namespace image	One snapshot
PanFS customers	65	PanFS/Lustre/GPFS	Namespace image	One snapshot
LANL clusters	6	NFS/GPFS	Namespace image	One snapshot
OpenCloud	1	HDFS	NameNode log	2 years
Altiscale	15	HDFS	NameNode log	1 day to 125 days
LinkedIn	1	HDFS	NameNode log	1 day
Yahoo!	1	HDFS	NameNode log	3 days
Cloudera	1	HDFS	NmaeNode log	2 days

Table 3.1: This data summarizes the specifics of the data collected from 96 file systems.

There are two types of workload traces: the namespace image and the metadata server (NameNode in HDFS) operation log. The namespace image is collected for all storage systems, and the operation log is only available from HDFS in Hadoop clusters. A file system namespace image (static information) is the one-day snapshot of file system namespace, and includes file size distribution, file counts in directories, and path depth. Operation logs are collected from Hadoop NameNodes during some period, which describes the runtime behavior of the entire file system. Several useful metrics can be derived from the operation log including percentage of different operations and locality in the pathnames. Based on these static or dynamic infor-

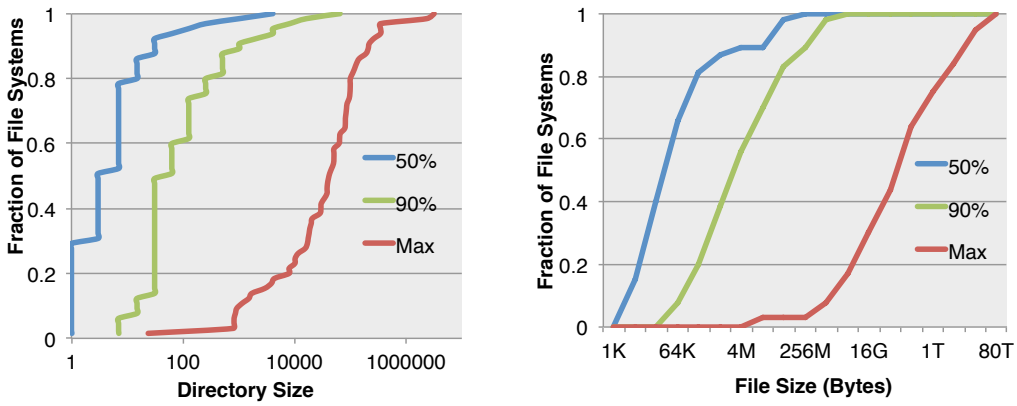


Figure 3.1: The distribution of 50th, 90th, and 100th percentile, and maximum directory sizes and file sizes of collected file system samples.

mation, we discuss possible design trade-offs that can be made to scale a metadata management system.

3.2 File System Namespace Statistics

To understand the structure of a file system namespace, this study looks at the statistical distribution of its many properties including directory size, file size, and entry depth. The resulting analysis shows that the old rule of thumb is still valid for today's file systems: most of these properties exhibit a skewed distribution.

Many file systems in the collected traces consist of lots of small files/directories and relatively few large files/directories. Figure 3.1 shows the distribution of the 50th percentile, 90th percentile, and 100th percentile directory/file size among all collected storage system traces. In about 60 of 70 file systems, more than 90% of directories are of a size smaller than 128KB. But a few very large directories that contain roughly millions of directory entries are also presented in these traces. File size has a similar distribution: nearly 81% traced file systems have median file size smaller than 256KB, and the largest file can be as large as a couple of terabytes. Their capacity is mostly consumed by large files.

A natural challenge for many file systems is to support efficient accesses to large directories. It is reported that many parallel computing applications are reported to require concurrent accesses to very large directories such as check pointing, and

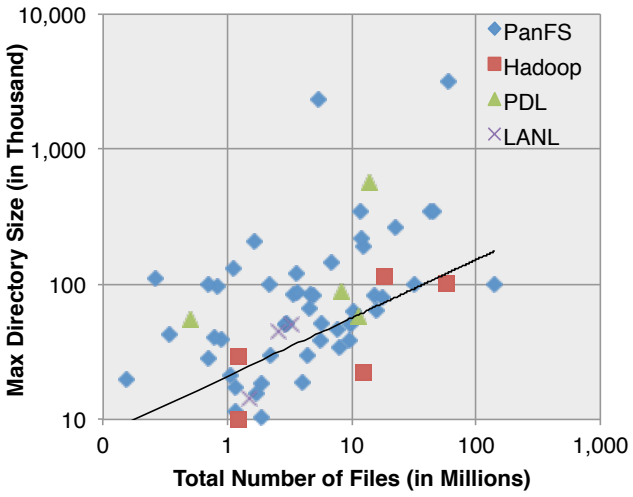


Figure 3.2: This scatter plot demonstrates the relation between the size of each file system sample and its maximum directory size. The black line is the trend lines that predicts the relation using linear regression.

map-reduce jobs [BGG+09, DG04]. To see how the large directories evolve when the file system scales, Figure 3.2 plots the relation between the file system size and its maximum directory size. Each data point is a file system sample, colored with different markers to label their sources. As we can see, the maximum directory size increases almost linearly as file system size grows. This implies that a scalable metadata service that meets future needs should support a large amount of small objects, as well as efficient distribution of large directories for concurrent accesses.

The shape of a file system tree has performance impacts on many metadata operations of which path resolution is an important step. For many file system designs, the number of components in a pathname is the number of lookup queries that need to be issued to the file system metadata cache or even to its on-disk data. Since there is no straightforward method to quantify or visualize the tree structure, we look at the distribution of the depth of each entry in the file system instead. Figure 3.3 shows the distribution of the entry depth per file system sample for a portion of the collected samples. (This information is not available for Panasas customer installations). We found that most file systems in the collected samples show 90% of their directories having a depth lower than 16, and 50% of directories having depth lower than 9.

It is also important to understand how the average depth of file entries grows as the file system becomes bigger. Figure 3.4 demonstrates the relationship between the file system size and the average depth of their entries. Both axes in this figure are shown in the log scale. It can be inferred from the figure that the average depth of the tree increases sub-linearly as the file system size grows. An obvious conjecture is that the average directory depth is proportional to the logarithm of the number of objects in the file system.

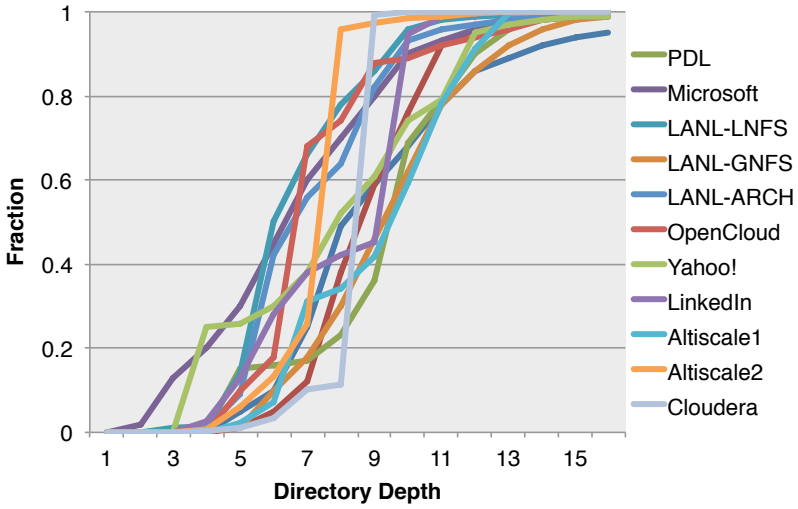


Figure 3.3: The distribution of directory depth of collected file systems traces. Many file systems have 50% of their directories with a depth lower than 9.

3.3 Dynamic Behaviors in Metadata Workload

One particularly important dynamic behavior of metadata workloads is how different types of metadata operations are executed during the period of log collection. Since only Hadoop clusters produce operation logs that contain the information about their dynamic behaviors, this section focuses the discussion on traces of the following organizations: Altiscale, LinkedIn, Yahoo!, Cloudera and OpenCloud. Most of these Hadoop clusters are of medium size at least, consisting of dozens of computer nodes.

To give a high-level overview about the dynamic behavior, let us first look at an overall summary of the frequency of each type of metadata operation. Figure 3.5 demonstrates the fraction of each type of metadata operation over the entire

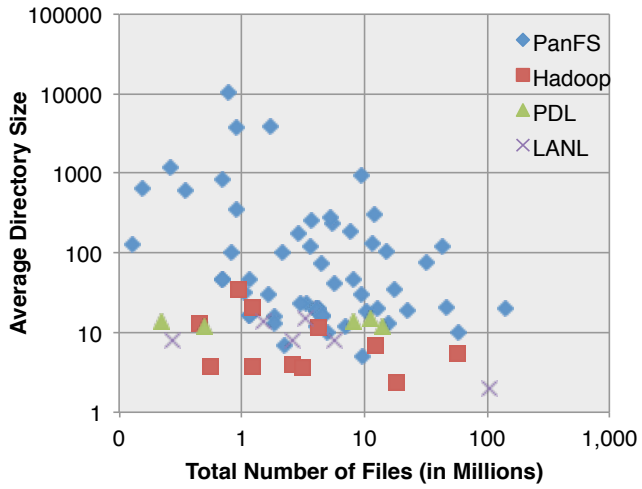


Figure 3.4: This scatter plot demonstrates the relationship between the size of each file system sample and the average depth of objects inside the system. The average depth of the tree increases sub-linearly as the file system size grows.

collection period. In all these traces, read-only operations (including `open`, `stat` and `readdir`) dominate the Hadoop workloads. (Please notice that `open` in Hadoop only opens the file for reading). `open` is the most prevalent metadata operation among most collected log traces. Other read-only metadata operations such as `stat` and `readdir` also make up a very large portion of the entire collected trace. This indicates that many Hadoop workloads are read-intensive. One typical example would be having lots of ETL jobs that extract a large amount of unstructured data that is transformed into data with proper format for future querying and analysis.

For Hadoop metadata workloads, update operations seem to be more prevalent than pure creation operations such as `create` and `mkdir`. Other complicated metadata operations such as `readdir` and `rename` consist of a small but visible fraction of the entire workload (from 1% to 5%). In some Hadoop applications, `readdir` is frequently used to list files in a directory or get status about a particular file to generate InputSplit data structure on job submission. `rename` is used more frequent than might be expected because Hadoop tasks use `rename` to achieve a simple form of transactions called output commit. The final output files are first written to temporary directories. Once the tasks finish successfully, temporary directories are renamed to the final output file names. This means that although the metadata service can

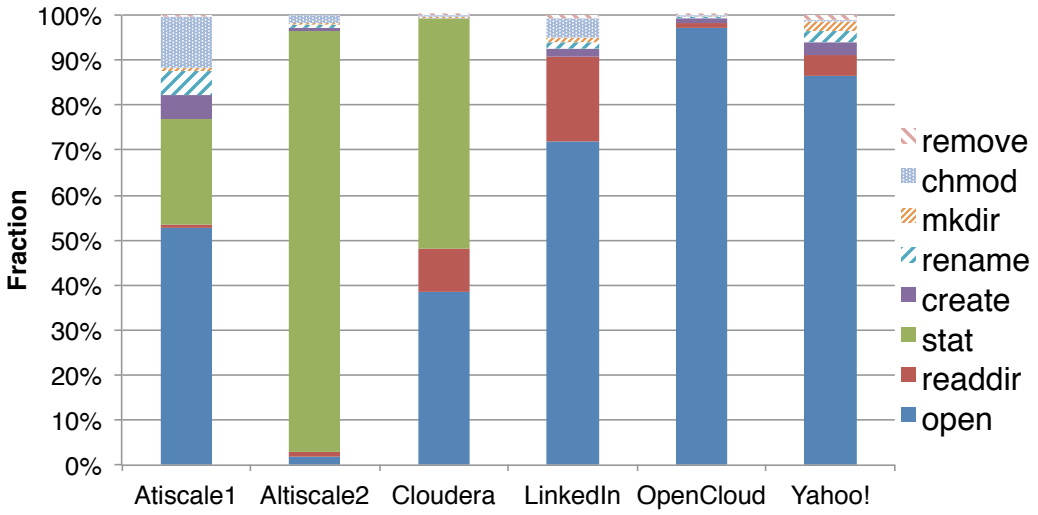


Figure 3.5: Distribution of file system operations in LinkedIn, Yahoo! and OpenCloud traces. Most of these workloads are read-intensive, but complicated operations like `rename` also have a significant presence.

be designed to favor those frequently used read-only point queries, the performance of these range operations should be also carefully taken into consideration.

To look into the detailed workload behaviors over time, two example file systems are selected: one is the one-day LinkedIn operation log, and the other is the 125-day Altiscale operation log. The one-day LinkedIn log trace has been used to evaluate the system prototype proposed by this thesis. The Altiscale trace shows the longest operation we have from a production Hadoop cluster. Figure 3.6 shows the change in fraction of each type of metadata operations over time for the LinkedIn cluster. Figure 3.7 presents the same information for the Altiscale cluster. The two figures also summarize the overall throughput of metadata operations every minute, depicted as a black solid line. In the LinkedIn trace, the variance of the distribution of metadata operations across the collection period is not as volatile as it appears in the Altiscale trace. For the LinkedIn trace, there are only a few spikes in the throughput during the whole period, which corresponds to the increase in the number of `readdir` operations. However, the Altiscale trace covers a much longer collection period than the LinkedIn trace. This implies that the distribution of metadata operations and

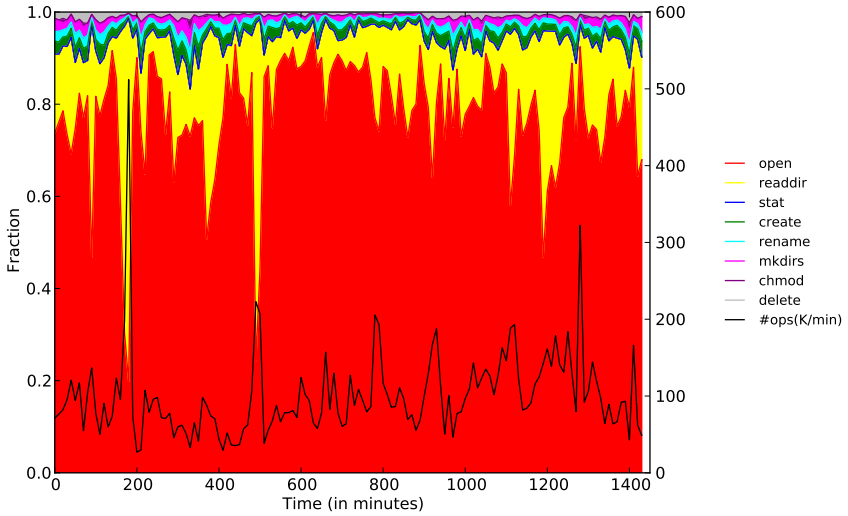


Figure 3.6: The distribution of each type of metadata operation in a one-day LinkedIn trace over time. The first (left) y-axis shows the fraction of each type of operation. The second (right) y-axis shows the total number of operations every minute in thousands. The variance of the distribution in the Altiscale cluster is higher than in the LinkedIn one.

access patterns may be stable for short periods, but will display drastic random changes if looked at for a longer period.

3.4 Statistical Properties of Metadata Operations

This sections looks into the access patterns of individual metadata operations. Figure 3.8 shows the depth of most pathnames accessed by each metadata operation in the LinkedIn trace. The depth is between 4 and 11. This result indicates that reducing iterative lookups for each pathname component can effectively enhance the performance of metadata operations, especially when file namespace is partitioned across metadata servers and the cost of each lookup is high.

The next step of the analysis is to determine whether inode caching can effectively improve path resolution; that is, to reduce the number of queries on an inode that go to the disk or remote servers. We discover that skews also exist in the path

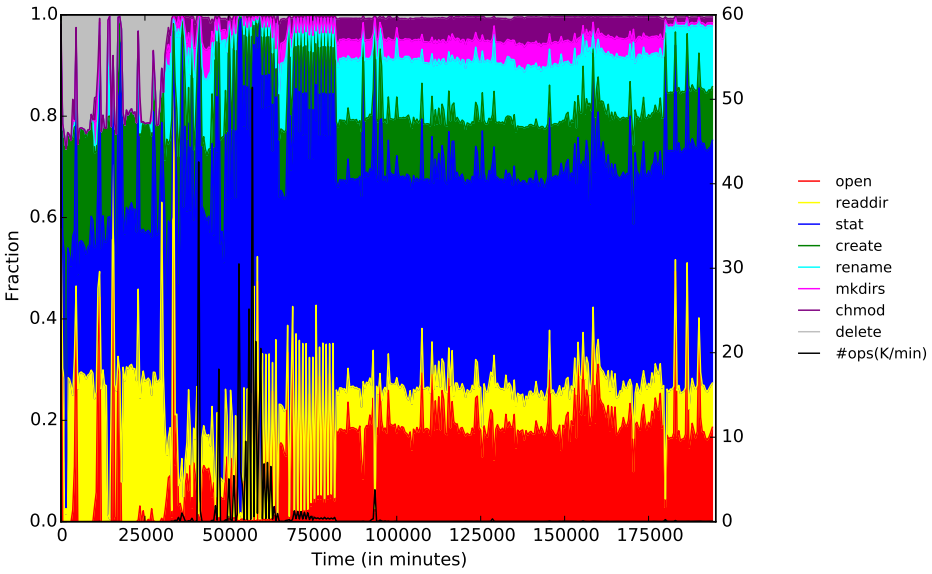


Figure 3.7: The distribution of each type of metadata operations in a 125-day Altiscale trace over time. The variance of the distribution in the Altiscale cluster is higher than in the LinkedIn one.

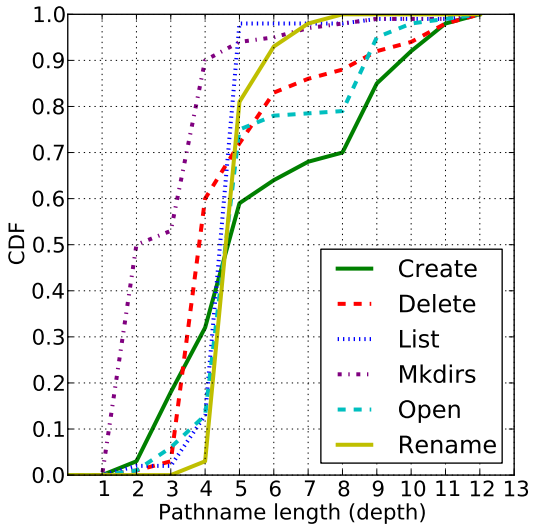


Figure 3.8: Distribution of metadata operations in the LinkedIn trace by length of accessed pathname.

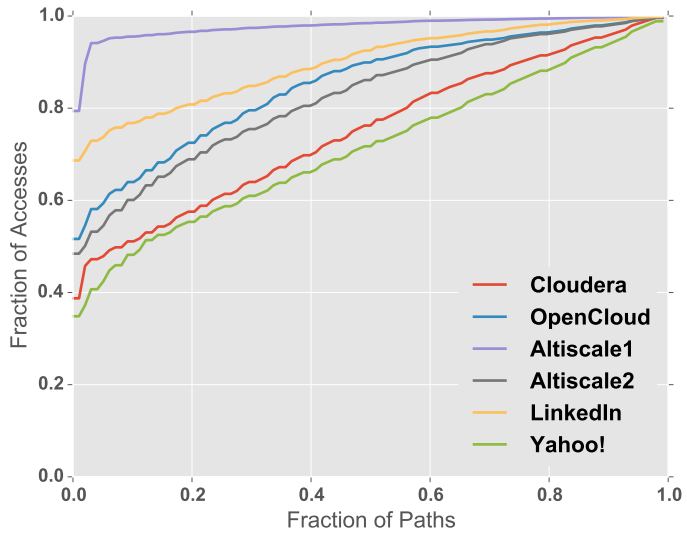


Figure 3.9: Distribution of access frequency of pathnames.

access patterns of all collected traces. Figure 3.9 shows the cumulative distribution of access frequencies of all pathnames in the collected traces. (Access frequency of a pathname counts every metadata operation performed on it.) Obviously, the access patterns are skewed toward a few frequently accessed pathnames. Since most metadata operations are read-only operations as shown in previous analysis, caching these pathnames can result in a very effective hit ratio, and directly benefit path resolution.

Since `rename` is one of the notoriously complicated metadata operations, especially for distributed file systems, an more detailed analysis of how `rename` is used in the metadata workloads is beneficial. As shown in previous analysis, there are a visible fraction (1% to 5%) of `rename` operations appearing in these workloads. A correct implementation of `rename` often requires acquiring multiple locks along the pathnames [DH06], which may limit its concurrency. We analyzed the concurrent `rename` operations appearing in the log traces to see what fraction of them actually generate conflicts. Concurrent operations are defined as those having time stamps recorded in the log that differ by no more than one second. Table 3.2 shows the fraction of `rename` operations having conflicts. The ratio is smaller than 0.1% for most of traces except Altiscale and Yahoo!. This means using simple locking mechanisms may be good enough for Hadoop-related workloads.

	Altiscale1	Altiscale2	Cloudera	LinkedIn	OpenCloud	Yahoo!
Num. Renames	1,232,768	5,078,108	25,299	1,874,109	810,002	2,704,961
Conflicts Ratio	0.04	0.0004	0.0015	0.0005	0.01	0.08

Table 3.2: Percentage of rename operations that generate conflicts in each collected trace.

3.5 Lessons from Workload Analysis

This section summarizes all the lessons and challenges learned from this workload study about designing a scalable metadata service for both local and distributed file systems.

- *Skewed Namespace Structure*: Typical file systems have skewed distribution in their directory size, file size and entry depth. It is a known challenge that file systems must handle lots of small directories and files on secondary devices like magnetic drives. This requires file systems to improve their external indexes to maintain locality for small directories and files. Moreover, file systems also need to provide efficient support for large directories and files. A direct way to improve accessing large directories and files in distributed file systems is to increase the access parallelism.

- *Diverse Metadata Workloads*: The metadata workloads that have been collected so far are very diverse, even for different periods in the same cluster. Read-only metadata operations such as `open`, `stat` and `readdir` are dominant operations, but other write operations such as `create`, `chmod` and `rename` are all a visible fraction of the operations that appeared in the collected Hadoop traces. Some HPC workloads also contain lots of applications that require fast parallel file creates for check-pointing [PG11]. Sometimes, slow implementation of `rename` may incur very high overhead [AT15]. This thesis is looking for a design that provides balanced performance for all types of metadata operations, in order to support a wide range of metadata workloads.

- *Skewed Accesses Patterns*: Accessing files and directories also appears to be skewed. One factor contributing to the skew is the file system's tree structure itself. Directories on the top of tree are more frequently accessed because the POSIX semantics require that permissions of each component are checked along the pathname. From the Hadoop trace analysis, we also found that some files

are more popular than others, which results in the imbalanced access patterns. It hints that ultilizing cache to avoid repeat accesses will enhance performance.

- *Writes with Few Conflicts*: The investigation into `rename` operations in Hadoop traces showed us that write-write conflicts are rare (fewer than 1% in many traces). This may be used to simplify the implementation of `rename` operations while still maintaining considerable performance.

Chapter 4

TableFS: A Stacked File System Design Layering on Key-Value Store

This chapter focuses on managing file system metadata in a single machine. Today's applications exhibit highly diverse I/O accessing patterns, which makes performance optimization of a general-purpose local file system a frustrating balancing act. Especially for file system metadata management, metadata are small in in size and often organized by the hierarchical file namespace, which often results in many small writes to the underlying storage devices. Examples include managing emails or thumbnail files, creating lock files for editing text files, or updating a file's atime. These I/O writing patterns are usually harmful to modern storage devices such as magnetic disks and solid state disks.

The core problem is that many conventional standard data structures used in previous file-systems including FFS and LFS optimize for one case at the expense of another. As discussed in the previous chapter, recent advances in write-optimized indexes such as Log-structured Merge (LSM) tree [OCGO96] are exciting because they have the potential to implement both efficient small writes and range scans which are necessary for file system metadata management. The key strength of these indexes is that they are featured high ingestion rate that are up to two orders of magnitude faster than traditional B-trees while matching or improving on the B-tree's point-query and range-query performance. Moreover, modern key-value stores that emphasize simple key-value interfaces and large in-memory caches have implemented these indexes as the core storage engine. Their implementations are "thin" enough to provide the performance levels required by file systems.

This chapter presents a modular file system design called TableFS that leverages write-optimized indexes and their modern key-value store implementation to manage file system metadata and tiny files. The modular design of TableFS provides two different schema that map VFS operations to a write-optimized index (key-value store), while it ensures locality and optimizes for different I/O workloads. LevelDB [Lev11] is used as the underlying key-value store for TableFS in our evaluation. LevelDB is an open-source implementation of LSM tree with extensive buffering and compact in-memory indexes. Our experiments results show that for workloads dominated by metadata and tiny files, this modular design backed by efficient key-value stores can improve the performance of the most modern local file systems in Linux by as much as an order of magnitude.

In the following sections, we will first give an overview about file system metadata and LSM-trees. We will then discuss the modular design and evaluation results of TableFS. Through the evaluation, we show that with the modular design, TableFS can be optimized to support workloads with different I/O characteristics on various modern storage devices.

4.1 Background

4.1.1 Analysis of File System Metadata Operations

The management of a file system namespace is complicated by the semantics and performance requirements of metadata operations. Figure 4.1 shows three types of metadata: file metadata, directory entries and directory metadata. File or directory metadata are attributes associated with a file or a directory such as permissions and timestamps. Directory entries are names and pointers stored in directories, which are used to index the file system namespace.

According to the number of objects involved in an operation, file system metadata operations can be categorized in the following three major types:

- *Point operation*: These operations access the metadata of a file or directory referenced by its pathname–for example, *open*, *stat* and *chmod*.

- *Range operation*: Range operations scan a list of entries with common properties. For example, *readdir* reads all the entries inside a directory.

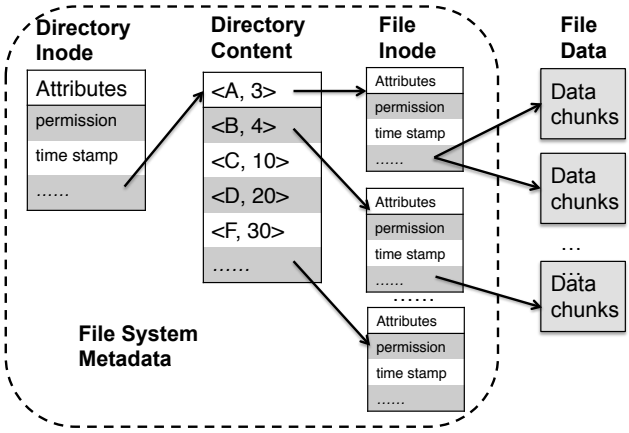


Figure 4.1: File and directory metadata structures

- *Tree operation*: An example of a tree operation is the *rename* operation. Renaming a directory to have a different pathname will change the structure of the file system namespace, influencing many other files or directories.

To represent file system metadata, a key-value store should at least support point queries such as `put`, `get` and `del`. Besides these basic queries, LevelDB used in TableFS provides relatively rich key-value APIs including range query (iterators over a range of entries) and batched writes, which can be used to implement directory readdir and atomic rename operations.

4.1.2 LSM-Tree and Its Implementation LevelDB

The log-structured merge tree (LSM-tree) is an write-optimized indexing data structure that manages a list of sorted key-value pairs on secondary storage devices, and supports *put, get* and *range scan* operations. An LSM-tree has four main components as shown in Figure 4.2:

- a *memtable* delays writing new and changed entries until it has a significant amount of changes to record in storage.
- a set of *immutable tables*, known as SSTables [CDG+06, Lev11], each essentially a static B-tree, storing a sorted list of entries in storage.
- a set of *in-memory indexes* including Bloom filters [Blo70] and key range mapping used for efficient lookup.

- a *compaction process* that re-organizes multiple SSTables by merge sort to delete stale entries, improve data sequentiality and decrease the number of SSTables a lookup might have to search.

Specifically, we discuss the design of LevelDB, a well-known open source implementation variant of LSM-tree [Lev11]. In LevelDB, newly inserted data is stored in the memtable and appended to a log file for failure recovery. When the total size of memtable exceeds a threshold (e.g., the default value is 4MB), the content of the memtable is spilled to disk. When a spill is triggered, dirty entries are sorted, indexed and written to disk as an SSTable. These entries may then be discarded from the memtable, and can be reloaded by searching each SSTable on disk, possibly stopping when the first match occurs if the SSTables are searched from most recent to oldest. The number of SSTables that need to be searched can be reduced by maintaining a Bloom filter [Blo70] on each, but with increasing numbers of records the disk access cost of finding a record not in memory increases.

Scan operations in LevelDB are used to find neighbor entries, or to iterate through all key-value pairs within a range. When performing a scan operation, LevelDB first searches each SSTable to place a cursor; it then increments cursors in multiple SSTables and merges key-value pairs into sorted order.

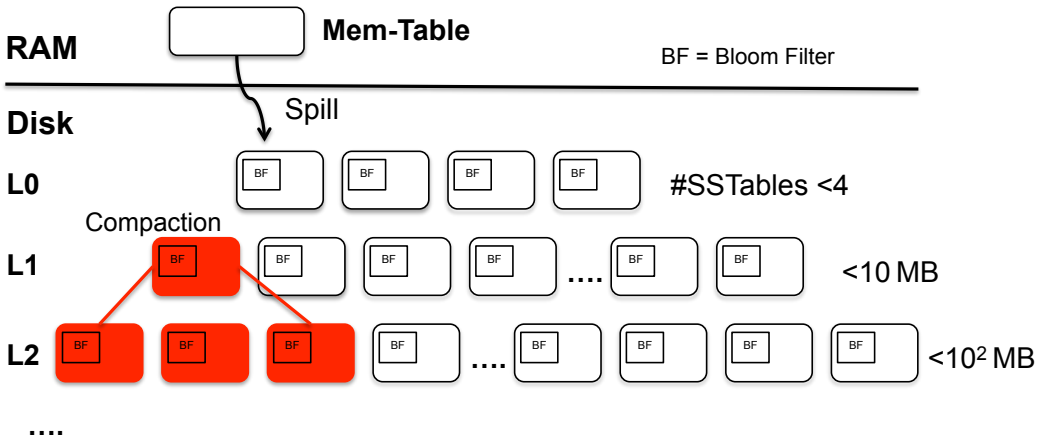


Figure 4.2: LevelDB represents data on disk in multiple SSTables that store sorted key-value pairs. Each solid rectangle represents an SSTable. LevelDB uses a multi-level structure to grow an LSM-tree store (with a growth factor $r = 10$).

A key difference between LevelDB (LSM-tree) and other B-tree index structures is its *level structure*. As illustrated in Figure 4.2, LevelDB extends the simple search all SSTables approach to further reduce read costs by dividing SSTables into sets, or levels. Levels are numbered starting from 0, and levels with a smaller number are referenced as "lower" levels. The 0th level of SSTables follows a simple formulation: each SSTable in this level may contain entries with any key/value, based on what was in memory at the time of its spill. LevelDB's SSTables in level $k+1$ are the results of compacting SSTables from level k ($k \geq 0$). In these higher levels, LevelDB maintains the following invariant: each SSTable is limited in size (2MB by default), and the key range spanning each SSTable is disjoint from the key range of all other SSTables at that level. Therefore querying for an entry in the higher levels only need to read at most one SSTable in each level. The total size of SSTables in each level follows a geometric progression: all SSTables have the same size and the maximum sum of the sizes of all SSTables at the level $k+1$ is r times larger than the sum of the sizes of all SSTables at previous level k. r is often referred as a "growth factor" that typically lies between 8 to 16. This ensures that the number of levels, that is, the maximum number of SSTables that need to be searched in the higher levels, grows logarithmically with increasing numbers of entries.

For levels other than level 0, when the aggregated size of SSTables in a level reaches the threshold, LevelDB picks an SSTable from that level and performs a compaction on it. SSTables are picked in a round-robin, incremental fashion within the key space for each level. When LevelDB decides to compact an SSTable at level k, it picks one, finds all other SSTables level $k+1$ that have an overlapping key range (and if $k = 0$ all the SSTables with an overlapping key range in level 0), and then merge sorts all of these SSTables, producing a set of bounded size SSTables with disjoint ranges at the next higher level. A newly produced SSTable cannot have more than a certain amount of overlapping data (e.g., 20MB) in the next level, which limits the future costs of any compaction procedure heuristically.

In summary, using a log-structured approach and a level structure, the amortized I/O cost per insertion or update is $O(\frac{1}{B} r log_r N)$, where B is the number of entries written in each block write, and N is the total number of unique keys in LevelDB. B can be quite large, reaching thousands in hard drive case. Compared to the I/O cost of a traditional B-tree – $O(log_B N)$, LSM-tree is much faster. The worst-case lookup incurs $O(log_r N)$ random I/O by searching SSTables in every non-zero level, assuming that finding an item in a level costs $O(1)$ random I/O. By adopting a memory index for each level, LSM-tree can retrieve key-value pairs using on average I/O cost $O(1)$, which is comparable to traditional B-trees.

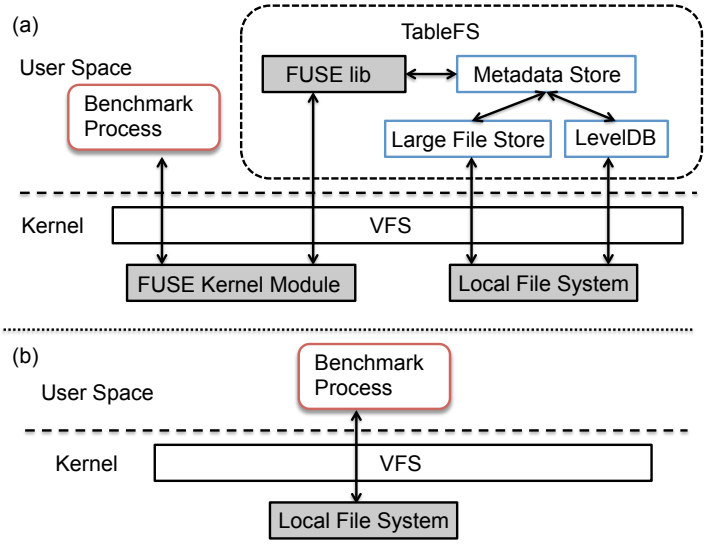


Figure 4.3: (a) The architecture of TableFS. A FUSE kernel module redirects file system calls from a benchmark process to TableFS, and TableFS stores objects into either LevelDB or a large file store. (b) When we benchmark a local file system, there is no FUSE overhead to be paid.

4.2 Design Overview of TableFS

As shown in Figure 4.3(a), TableFS uses a modular design that leverages only simple APIs provided by the underlying key-value stores such as PUT, GET, etc. TableFS represents directories, inodes and small files in one all-encompassing table in the key-value stores. And the key-value store compacts these small objects into large objects, and only writes large objects (such as write-ahead logs, SSTables, and large files) to the local disk. For fast prototyping, TableFS exploits the FUSE user level file system infrastructure to interpose itself on top of the local file system.

4.2.1 Local File System as Object Store

There is no explicit space management in TableFS. Instead, it uses the local file system for allocation and storage of objects. Because TableFS packs directories, inodes and small files into a key-value table, and the key-value store (as LevelDB used in this implementation) keeps sorted logs (SSTables) of about 2MB each, the

local file system sees many fewer, larger objects. We use Ext4 as the object store for TableFS in all experiments.

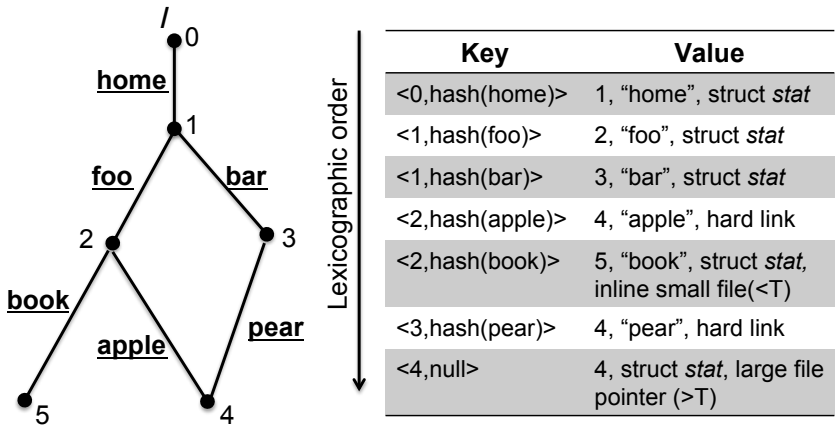
Files larger than T bytes are stored directly in the object store named according to their inode number. The object store uses a two-level directory tree in the local file system, storing a file with inode number I as "/LargeFileStore/K/J/I" where $J = I \div 10000$ and $K = J \div 10000$. This is to circumvent any scalability limits on directory entries in the underlying local file systems. In TableFS today, the threshold for blobbing a file T is 4KB, which is the median size of files in desktop workloads [MB11], although others have suggested T be at least 256KB and perhaps as large as 1MB [SIG07].

The rationale of packing small objects into write-optimized key-value store and storing files separately based on size is to avoid unnecessary random I/Os for small writes. We then show that by carefully picking table schema for metadata and small files, fast read performance can also be achieved.

4.2.2 Table Schema

TableFS's metadata store aggregates directory entries, inode attributes and small files into one key-value table with a row for each file. To link together the hierarchical structure of the user's namespace, the rows of the table are ordered by a variable-length key consisting of the 64-bit inode number of a file's parent directory and its filename string (final component of its pathname). The value of a row contains inode attributes, such as inode number, ownership, access mode, file size and timestamps (*struct stat* in Linux). For small files, the file's row also contains the file's data. Figure 4.4 shows an example of storing a sample file system's metadata into one key-value table.

To resolve a single pathname, TableFS starts searching from the root inode, which has a well-known inode number (0). Traversing the user's directory tree involves constructing a search key by concatenating the inode number of the current directory with the hash of next component name in the pathname. Unlike Btrfs [RBM12], TableFS does not need the second version of each directory entry because the entire set of attributes are returned in the *readdir* scan. All entries in the same directory have rows that share the same first 64 bits of their table key. For *readdir* operations, once the inode number of the target directory has been retrieved, a scan sequentially lists all entries having the directory's inode number as the first 64 bits of their table key.



Key	Value
<0,hash(home)>	1, “home”, struct *stat*
<1,hash(foo)>	2, “foo”, struct *stat*
<1,hash(bar)>	3, “bar”, struct *stat*
<2,hash(apple)>	4, “apple”, hard link
<2,hash(book)>	5, “book”, struct *stat,* inline small file(<T)
<3,hash(pear)>	4, “pear”, hard link
<4,null>	4, struct *stat*, large file pointer (>T)

Figure 4.4: An example illustrates table schema used by TableFS's metadata store. The file with inode number 4 has two hard links, one called “apple” from directory *foo* and the other called “pear” from directory *bar*.

There are other possible table schemas. One uses a full path as the primary key to index all files and directories [WJea15]. Its drawback is that it requires a read-modify-write of lots of rows when renaming a directory. As shown in Chapter 3, `rename` is not an uncommon operation, especially in Hadoop workload where it is used as transaction primitive for protecting the integrity of results generated by Hadoop jobs. Another schema uses the inode number as the primary key [AT15], and stores all directory entries in the value field of a directory row. When inserting a new entry, a read-modify-write of the entire row and the addition of another row are required, which increases the write amplification. This helps with the `readdir` operation because all directory entries are stored even closer both logically and physically to each other in the sense that one lookup retrieving all directory entries in a directory.

4.2.3 Hard Links

Hard links, as usual, are a special case because two or more rows must have the same inode attributes and data. Whenever TableFS creates the second hard link to a file, it creates a separate row for the file itself, with a null name, and its own inode number as its parent's inode number in the row key. As illustrated in Figure 4.4, creating a hard link also modifies the directory entry such that each row naming the

file has an attribute indicating the directory entry as a hard link to the file object's inode row.

4.2.4 Scan Operation Optimization

TableFS utilizes the scan operation provided by LevelDB to implement *readdir()* system calls. The scan operation in LevelDB is designed to support iteration over arbitrary key ranges, which may require searching SSTables at each level. In such a case, Bloom filters cannot help to reduce the number of SSTables to search. However, in TableFS, *readdir()* only scans keys sharing a common prefix — the inode number of the parent directory. For each SSTable, an additional Bloom filter is maintained by TableFS, to keep track of all inode numbers that appear as the first 64 bits of row keys in the SSTable. This is a parent inode Bloom Filter. Before starting an iterator in an SSTable for *readdir()*, TableFS can first check its parent inode Bloom filter to find out whether it contains information on any of the desired directory entries. Therefore, unnecessary iterations over SSTables that do not contain any of the requested directory entries can be avoided.

4.2.5 Inode Number Allocation

TableFS uses a global counter for allocating inode numbers. The counter increments when creating a new file or a new directory. Since we use 64-bit inode numbers, it will not soon be necessary to recycle the inode number of deleted entries. Coping with operating systems that use 32 bit inode numbers may require frequent inode number recycling, a problem beyond the scope of this thesis and addressed by many file systems.

4.2.6 Concurrency Control

We leverage the atomic insertion of a batch of writes provided by LevelDB to implement *rename* operation. The atomic batch write guarantees that a sequence of updates to the database are applied in order, and committed to the write-ahead log atomically. Thus the *rename* operation can be implemented as a batch of two operations: insert the new directory entry and delete the stale entry. However, LevelDB does not support atomic row read-modify-write operations. For operations like *chmod* and *utime*, since all of an inode's attributes are stored in a single key-

value pair, TableFS must read-modify-write attributes atomically. In the TableFS core layer, we implement per-inode locking mechanism to ensure correctness under concurrent access.

4.2.7 Journaling

TableFS relies on LevelDB and the local file system to achieve journaling. LevelDB has its own write-ahead log that journals all updates to the table. LevelDB can be set to commit the log to disk synchronously or asynchronously. To achieve a consistency guarantee similar to "ordered mode" in Ext4, TableFS forces LevelDB to commit the write-ahead log to disk periodically (by default it is committed every 5 seconds).

4.2.8 Column-Style Table for Faster Insertion

Some applications, such as checkpointing, prefer fast insertion performance or fast pathname lookup rather than fast directory list performance. Moreover, solid state disk has asymmetric performance for reads and writes, and have limitation for number of erases in its life cycle which means limited writes. To better support such applications and storage devices, TableFS supports an alternative metadata table schema, called *column-style*, that speeds up the throughput of insertion, modification, single-entry lookup, and significantly reduces write amplifications. By using this second smaller table for the most important operations, TableFS can disable compaction of the full metadata table, which reduces write amplification at the expense of worse random read performance.

As shown in Figure 4.5, TableFS's column-style schema adds a second index table sorted on the same key, which stores only the final pathname component string, permissions and a pointer (to the most recent corresponding record in the full metadata table). Like a secondary index, this table is smaller than the full table, so it caches better and its compactions are less frequent. It can satisfy `lookup` and `readdir` operations, the most important non-mutation metadata accesses, without dereferencing the pointer. But it cannot satisfy `stat` and `read` without one more SSTable reference. TableFS eliminates compaction in the full table (rarely, if ever, compacting the full table). This speeds up insertion intensive workloads significantly. Moreover, because the index table contains a pointer (log ID and offset in the appropriate log file), and because each mutation of a directory entry or its embedded data rewrites the entire row of the full table, there will only be one disk read if a non-mutation access is not satisfied in the index table, speeding up single file metadata accesses that miss in

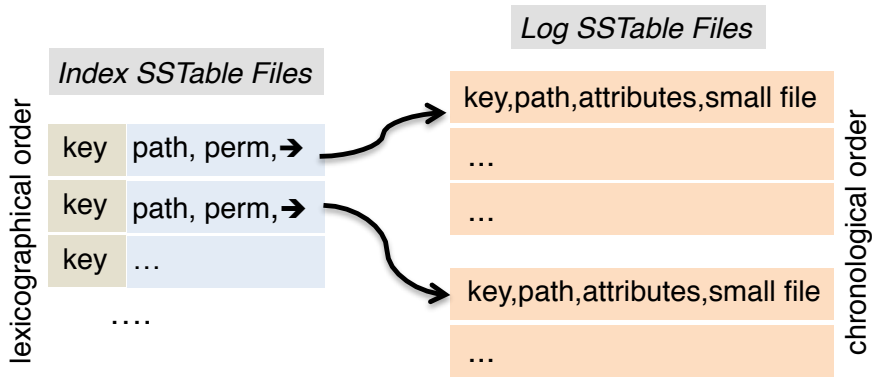


Figure 4.5: Column-style stores index and log tables separately. Index tables contain frequently accessed attributes for file lookups and a pointer to the location of full file metadata in the most recent log file. Index tables are normally compacted while log tables are rarely or never compacted, reducing the total work for TableFS.

cache relative to the standard LevelDB multiple level search. The disadvantage of this approach is that the full table, as a collection of uncompacted log files, will not be in sorted order on disk, so scans that cannot be satisfied in the index table will be more expensive, and will not be reorganized frequently by compaction. Cleaning rows no longer referenced in the full table and resorting by primary key (if needed at all) can be done by a background defragmentation service (a variation on compaction). In many cases, the cleaning procedue can be scheduled very infrequenlty, or even never scheduled. As shown in previous studies [WN13, MB11], most storage capacity in normal file systems are used to store the file data from large files. Epscially for scrath file systems for storing check-pointing files, metadata information about these files are hardly changed after their initial generation. A recent work WiscKey [LPG+17] shows that while having garbage collection of log files in the background the random write throughput of column-style store descreases by at most 35%, and is still 70 times faster than the original LevelDB solution.

4.2.9 TableFS in the Kernel

A kernel-native TableFS file system is a stacked file system, similar to eCryptfs [Hal05], treating a second local file system as an object store, as shown in Figure 4.6(a). An implementation of a Log-Structured Merge (LSM) tree [OCGO96] used

for storing TableFS in the associated object store, such as LevelDB [Lev11], is likely to have an asynchronous compaction thread that is more conveniently executed at the user level in a TableFS daemon, as illustrated in Figure 4.6(b).

For the experiments in this paper, we bracket the performance of a kernel-native TableFS (Figure 4.6(a)), between a FUSE-based user-level TableFS (Figure 4.6(b)) and an application-embedded user-level library TableFS (Figure 4.6(c)). In our experiment, there is no TableFS function in the kernel; all of TableFS resides in the user level FUSE daemon or an application-embedded TableFS library, illustrated in Figure 4.6(c).

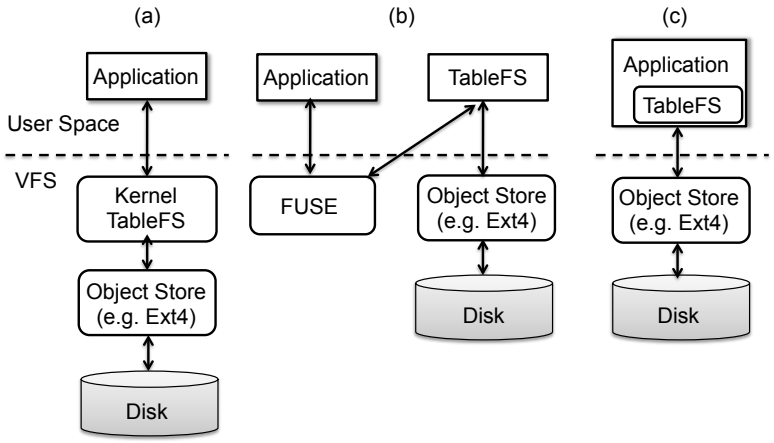


Figure 4.6: Three different implementations of TableFS: (a) the kernel-native TableFS, (b) the FUSE version of TableFS, and (c) the library version of TableFS. In the following evaluation section, (b) and (c) are presented to bracket the performance of (a), which was not implemented.

TableFS entirely at user-level in a FUSE daemon is unfairly slow because of the excess kernel crossings and scheduling delays experienced by FUSE file systems [BGG+09, SSM+13]. TableFS embedded entirely in the benchmark application as a library is not sharable, and unrealistically fast because of the infrequency of system calls. We approximate the performance of a kernel-native TableFS using the library version and preceding each reference to the TableFS library with a write("*/dev/null*", N bytes) to account for the system call and data transfer overhead. N is chosen to match the size of data passed through each system call.

4.3 Evaluation

4.3.1 Evaluation System

We evaluate our TableFS prototype on Linux desktop computers equipped as follows:

Linux	Ubuntu 12.10, Kernel 3.6.6 64-bit version
CPU	AMD Opteron Processor 242 Dual Core
DRAM	16GB DDR SDRAM
Hard Disk	Western Digital WD2001FASS-00U0B0 SATA, 7200rpm, 2TB Random Seeks 100 seeks/sec peak Sequential Reads 137.6 MB/sec peak Sequential Writes 135.4 MB/sec peak

We compare TableFS with Linux's most sophisticated local file systems: Ext4, XFS, and Btrfs (used the default versions that come with Linux kernel 3.6.6). Ext4 is mounted with "ordered" journaling to force all data to be flushed out to disk before its metadata is committed to disk. By default, Ext4's journal is asynchronously committed to disks every 5 seconds. XFS and Btrfs use similar policies to asynchronously update journals. Btrfs, by default, duplicates metadata and calculates checksums for data and metadata. We disable both features (unavailable in the other file systems) when benchmarking Btrfs to avoid penalizing it. Since the tested filesystems have different inode sizes (Ext4 and XFS use 256 bytes and Btrfs uses 136 bytes), we pessimistically penalize TableFS by padding its inode attributes to 256 bytes. This slows down TableFS's ability to process metadata-intensive workloads significantly, but it still performs quite well. In some benchmarks, we also changed the Linux boot parameters to limit the machines' available memory below certain threshold, in order to ensure out-of-RAM performance.

4.3.2 Data-Intensive Macrobenchmark

We run two sets of macrobenchmarks on the FUSE version of TableFS, which provides a full featured, transparent application service. Instead of using a metadata-

intensive workload, emphasized in the previous and later sections of this chapter, we emphasize data-intensive workload. Our goal is to demonstrate that TableFS is capable of reasonable performance for the traditional workloads that are often used to test local file systems.

Kernel build is a macrobenchmark that uses a Linux kernel compilation and related operations to compare TableFS's performance to the other tested file systems. In the kernel build test, we use the Linux 3.0.1 source tree (whose compressed tar archive is about 73 MB in size). In this test, we run four operations in this order:

- **untar**: untar the source tarball;
- **grep**: grep "nonexistent pattern" over all of the source tree;
- **make**: run *make* inside the source tree;
- **gzip**: gzip the entire source tree.

After compilation, the source tree contains 45,567 files with a total size of 551MB. The machine's available memory is set to be 350MB, and therefore compiled data are forced to be written to the disk.

Figure 4.7 shows the average runtime of three runs of these four macro-benchmarks using Ext4, XFS, Btrfs and TableFS-FUSE. (The variances across three runs for benchmarks are all smaller than 1%). For each macro-benchmark, the runtime is normalized by dividing by the minimum value. Summing the operations, TableFS-FUSE is slowed by about 20%, but it is also incurring significant overhead since it moves all data through the user-level FUSE daemon and the kernel twice, instead of only through the kernel once, as illustrated in Figure 4.6. Table 4.7 also shows that the degraded performance of Ext4, XFS, and Btrfs when they are accessed through FUSE is about the same as TableFS-FUSE.

Postmark was designed to measure the performance of a file system used for e-mail, and web based services [Kat97]. It creates a large number of small randomly-sized files between 512B and 4KB, performs a specified number of transactions on them, and then deletes all of them. Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append. The configuration used for these experiments consists of two million transactions on one million files, and the biases for transaction types are equal. The experiments were run with the available memory set to be 1400 MB, too small to fit the entire datasets (about 3GB) in memory.

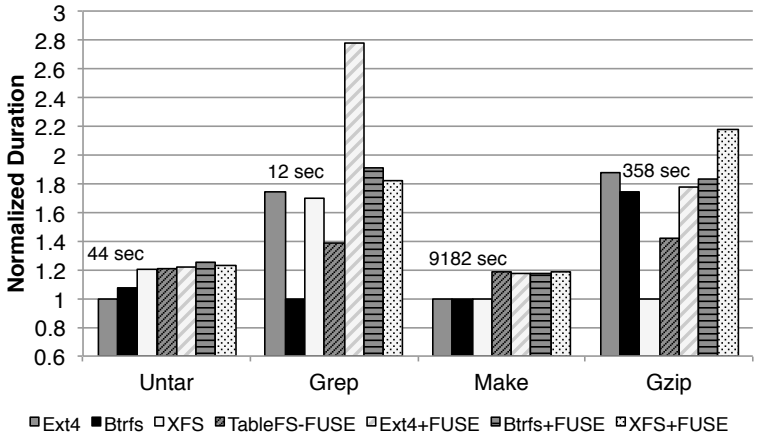


Figure 4.7: The normalized elapsed time for kernel building. All elapsed time is divided by the minimum value (1.0 bar). The legends above each bar show the minimum value in seconds.

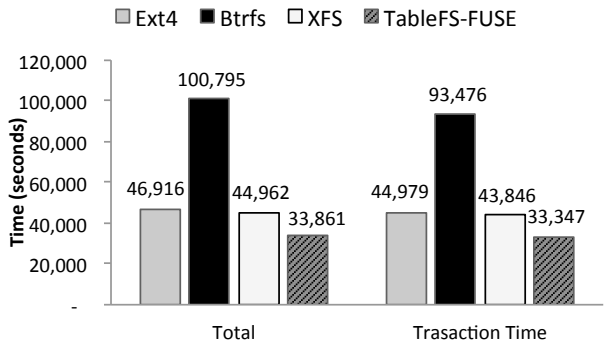


Figure 4.8: The elapsed time for both the entire run of Postmark and the transactions phase of Postmark for the four tested file systems.

Figure 4.8 shows the Postmark results for the four tested file systems. TableFS outperforms other tested file systems by at least 23% during the transctions phase. Figure 4.9 gives the average throughput of each type of operations individually. TableFS runs faster than the other tested filesystems for *read*, *append* and *deletion*, but runs slower for the *creation*. In Postmark, the *creation* phase creates files in the alphabatical order of their filenames. Thus the *creation* phase is a sequential insertion workload for all file systems, and Ext4 and XFS perform very efficiently in this workload. Since the size of created files are all smaller than 4KB, TableFS

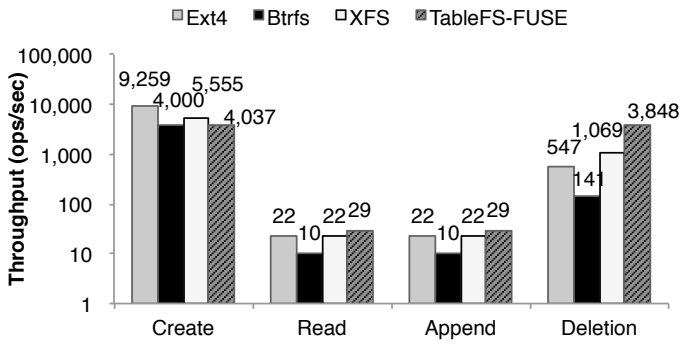


Figure 4.9: Average throughput of each type of operation in the Postmark benchmark.

will pack file data with metadata into one row stored in LevelDB. Without using column-style, TableFS-FUSE pays for the overhead from FUSE and writing file data twice due to LevelDB's journaling approach: The first time LevelDB writes it to the write-ahead log, and the second time to an SSTable during compaction.

4.3.3 TableFS-FUSE Overhead Analysis

To understand the overhead of FUSE in TableFS-FUSE, and estimate the performance of an in-kernel TableFS, we ran a micro-benchmark against TableFS-FUSE and TableFS-Library ((b) and (c) in Figure 4.6). This micro-benchmark creates one million zero-length files in one directory starting with an empty file system. The amount of memory available to the evaluation system is 1400 MB, almost enough to fit the benchmark in memory. But durability config forces data to disk so only negative lookups are faster with a larger cache.

Figure 4.10 shows the total runtime of the experiment. TableFS-FUSE is about 3 times slower than TableFS-Libary.

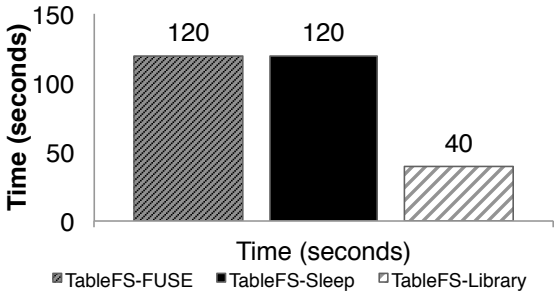


Figure 4.10: The elapsed time for creating 1M zero-length files on three versions of TableFS.

Figure 4.11 shows the total disk traffic gathered from the Linux proc file system (*/proc/diskstats*) during the test. Relative to TableFS-Library, TableFS-FUSE writes almost as twice as many bytes to the disk, and reads almost 100 times as much. This additional disk traffic originates from two sources: 1) under a slower insertion rate, LevelDB tends to compact more often as we will explain below; and 2) the FUSE framework populates the kernel's cache with its own version of inodes, competing with the local file system for cache memory.

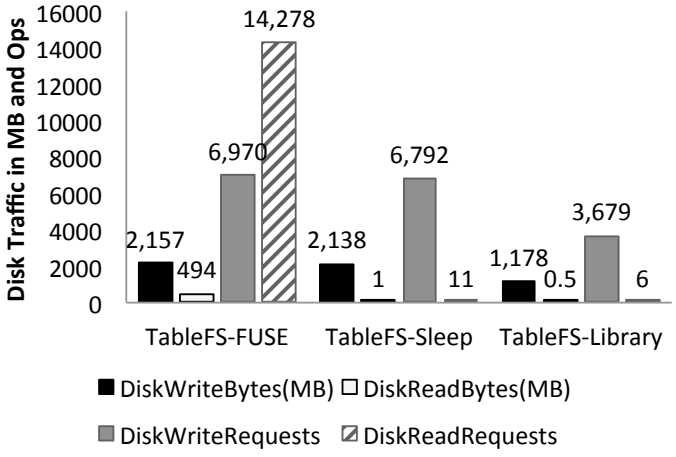
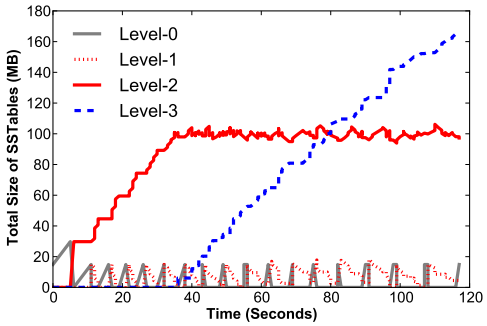


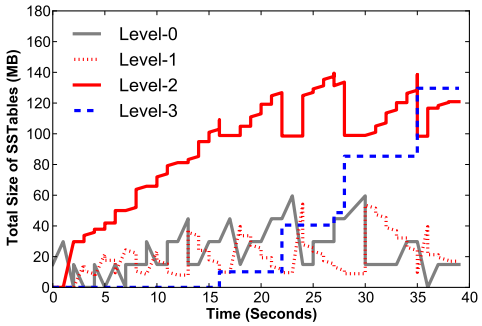
Figure 4.11: Total disk traffic associated with Figure 4.10

To illustrate the first point, we show LevelDB's compaction process during this test in Figure 4.12. Figure 4.12 shows the total size of SSTables in each Level over time. The compaction process will move SSTables from one level to the next. For

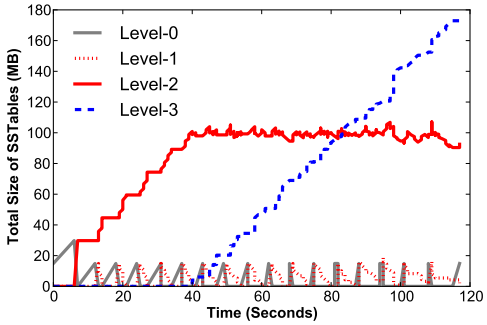
each compaction in Level 0, LevelDB will compact all SSTables with overlapping ranges (which in this benchmark will be almost all SSTables in levels 0 and 1). At the end of a compaction, the next compaction will repeat similar work, except the number of level 0 SSTables will be proportional to the data insertion rate. For each compaction, if the insertion rate is slower (Figure 4.12(a)), compaction in Level 0 finds fewer overlapping SSTables than TableFS-Library (Figure 4.12(b)). In Figure 4.12(b), the level 0 size (blue line) exceeds 20MB for much of the test, while in 4.12(a) it never exceeds 20MB after the first compaction. Therefore, LevelDB performs more compactions to integrate the same arriving log of changes when insertion is slower.



(a) TableFS-FUSE



(b) TableFS-Library



(c) TableFS-Sleep

Figure 4.12: Changes of total size of SSTables in each level over time during the creation of 1M zero-length files for three TableFS models. TableFS-Sleep illustrates similar compaction behavior similar to TableFS-FUSE.

To negate the different levels of compaction work, we deliberately slow down TableFS-Library to run at the same speed as TableFS-FUSE by adding *sleep 80ms* ev-

ery 1000 operations (80ms was empirically derived to match the run time of TableFS-FUSE). This model of TableFS is called TableFS-Sleep and is shown in Figure 4.11 and 4.12 (c). TableFS-Sleep causes almost the same pattern of compactions as does TableFS-FUSE and induces about the same write traffic (Figure 4.11). But unlike TableFS-FUSE, TableFS-Sleep can use more of the kernel page cache to store SSTables than TableFS-FUSE. Thus, as shown in Figure 4.11, TableFS-Sleep writes the same amount of data as TableFS-FUSE but does much less disk reading.

To estimate TableFS performance without FUSE overhead, we use TableFS-Library to avoid double caching, and perform a *write*("*/dev/null*", *N* bytes) on every TableFS invocation to model the kernel's system call and argument data transfer overhead. This model is called TableFS-Predict and is used in the following sections to predict metadata efficiency of a kernel TableFS.

4.3.4 Metadata-Intensive Microbenchmark

Metadata-only Benchmark

In this section, our goal is to measure the efficiency of pure metadata operations through four micro-benchmarks. Each micro-benchmark consists of two phases: a) create and b) test. For all four tests, the create phase is the same:

- a) *create*: In "create", the benchmark application generates directories in depth first order, and then creates one million zero-length files in the appropriate parent directories in a random order, according to a realistic or synthesized namespace.

The test phases in the benchmark are:

- b1) *null*: In test 1, the test phase is null because create is what we are measuring.
- b2) *query*: This workload issues one million read or write queries to random (uniform) files or directories. A read query calls *stat* on the file, and a write query randomly does either a *chmod* or *utime* to update the mode or the timestamp attributes.
- b3) *rename*: This workload issues a half million rename operations to random (uniform) files, moving the file to another randomly chosen directory.

- b4) *delete*: This workload issues a half million delete operations to randomly chosen files.

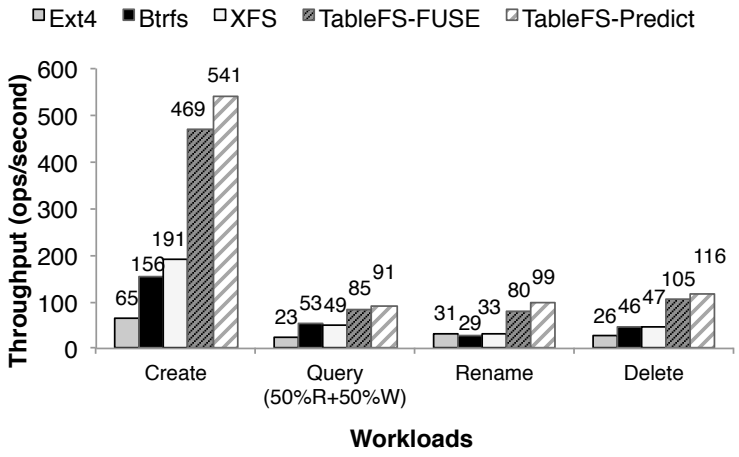


Figure 4.13: Average throughput during four different workloads for five tested systems. All tests were run for three times, and the coefficient of variation was less than 1%.

The captured file system namespace used in the experiment was taken from a personal Ubuntu desktop of a research team member. There were 172,252 directories, each with 11 files on average, and the average depth of the namespace is 8 directories. We also used the Impressions tool [AADAD09] to generate a "standard namespace". This synthetic namespace yields similar results, so its data is omitted from this paper. Between the create and test phase of each run, we unmount and re-mount local filesystems to clear kernel caches. To test out-of-RAM performance, we limit the machine's available memory to 350MB which does not fit the entire test in memory.

In Figure 4.13, the create results are from the null test. The workload in the create phase does random file insertion, which generates more pressure to the underlying file system and is different from the create phase used in Section 4.3.2. The other test results do not include the create phase, which is the same across all tests. Both TableFS-Predict and TableFS-FUSE runs are almost 2 to 3 times faster than the other local file systems in all tests.

Figure 4.14 shows the total number of disk read and write requests during the query workload, the test in which TableFS has the least advantage. Both versions of TableFS issue many fewer disk writes, effectively aggregating changes into larger

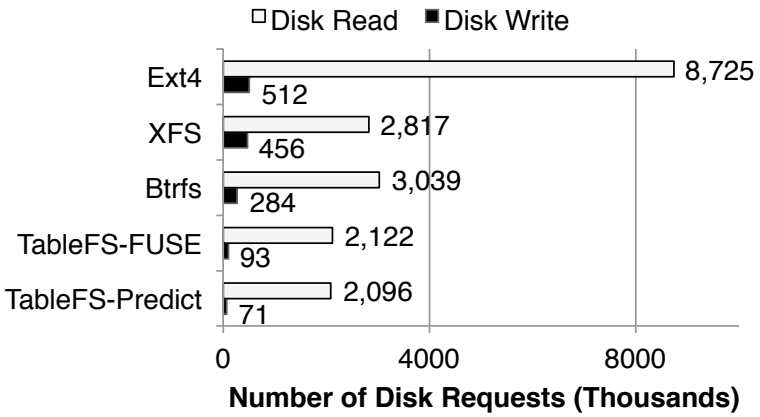


Figure 4.14: Total number of disk read/write requests during 50%Read+50%Write query workload for five tested systems.

sequential writes. For read requests, because of bloom filtering and in-memory indexing, TableFS issues fewer read requests. Therefore TableFS's total number of disk requests is smaller than the other tested file systems.

Scan Queries

In addition to point queries such as *stat, chmod* and *utime*, range queries such as *readdir* are important metadata operations. To test the performance of *readdir*, we modify the micro-benchmark to perform multiple *readdir* operations in the generated directory tree. To show the trade-offs involved in embedding small files, we create 1KB files (with random data) instead of zero byte files. For the test phase, we use the following three operations:

- b5) *readdir*: The benchmark application performs *readdir()* on 100,000 randomly picked directories.

- b6) *readdir+stat*: The benchmark application performs *readdir()* on 100,000 randomly picked directories, and for each returned directory entry, performs a *stat* operation. This simulates "ls -l".

- b7) *readdir+read*: Similar to *readdir+stat*, but for each returned directory entry, it reads the entire file (if returned entry is a file) instead of *stat*.

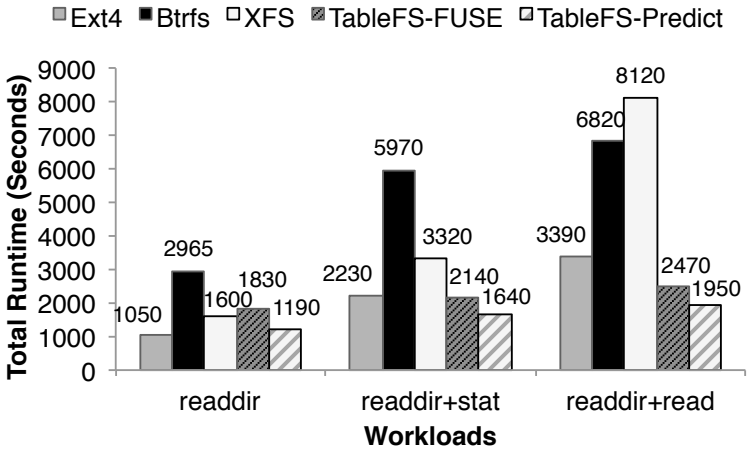


Figure 4.15: Total run-time of three *readdir* workloads for five tested file systems.

Figure 4.15 shows the total time needed to complete each *readdir* workload (the average of three runs). In the pure *readdir* workload, TableFS-Predict is slower than Ext4 because of read amplification, that is, for each *readdir* operation, TableFS fetches directory entries along with unnecessary inode attributes and file data. However, in the other two workloads when at least one of the attributes or file data is needed, TableFS is faster than Ext4, XFS, and Btrfs, since many random disk accesses are avoided by embedding inodes and small files.

Benchmark with Larger Directories

Because the scalability of small files is of topical interest [Whe10], we modified the zero-byte file create phase to create 100 million files (a number of files rarely seen in a local file system today). In this benchmark, we allowed the memory available to the evaluation system to be the full 16GB of physical memory.

Figure 4.16 shows a timeline of the creation rate for four file systems. In the beginning of this test, there is a throughput spike that is caused by everything fitting in the cache. Later in the test, the creation rate of all tested file systems slows down because the non-existence test in each create is applied to ever larger on-disk data structures. Btrfs suffers the most serious drop, slowing down to 100 operations per second at some points. TableFS-FUSE maintains more steady performance with an average speed of more than 2,200 operations per second and *is 10 times faster than all other tested file systems.*

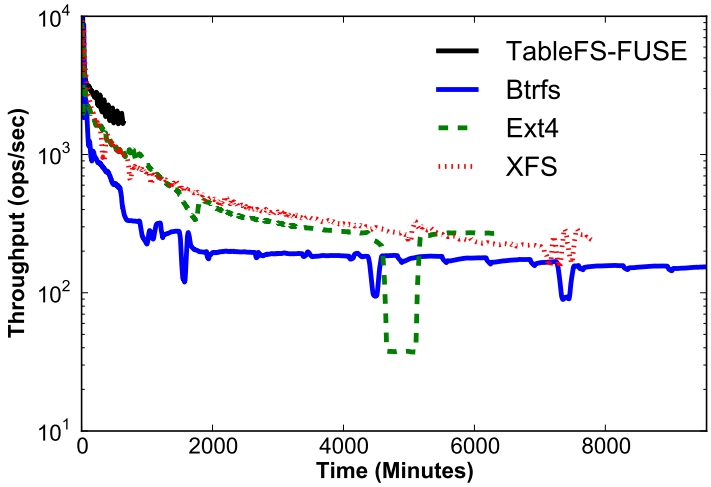


Figure 4.16: Throughput of all four tested file systems while creating 100 million zero-length files. TableFS-FUSE is almost 10× faster than the other tested file systems in the later stage of this experiment. The data is sampled in every 10 seconds and smoothed over 100 seconds. The vertical axis is shown on a log scale.

All tested file systems have throughput fluctuations during the test. This kind of fluctuation might be caused by on disk data structure maintenance. In TableFS, this behavior is caused by compactions in LevelDB, in which SSTables are merged and sequentially written back to disk.

Solid State Drive Results

We applied the "create-query" microbenchmark described in previous section to a 120GB SATA II 2.5in Intel 520 Solid State Drive (SSD). Random read throughput is 5,000 IO/s at peak, and random write throughput peaks at 2,500 IO/s. Sequential read throughput peaks at 245MB/sec, and sequential write throughput peaks at 107MB/sec. Btrfs has an "ssd" optimization mount option which we enabled.

Figure 4.17 shows the throughput averaged over three runs of the create and query phases. In comparison to Figure 4.13, all results are about 10 times faster. The performance of TableFS-Predict is comparable to the fastest. Figure 4.18 shows the total number of disk requests and disk bytes moved during the query phase. While TableFS achieves fewer disk writes that helps with the life span of solid state disks, it reads much more data from SSD than XFS and Btfs because of compaction

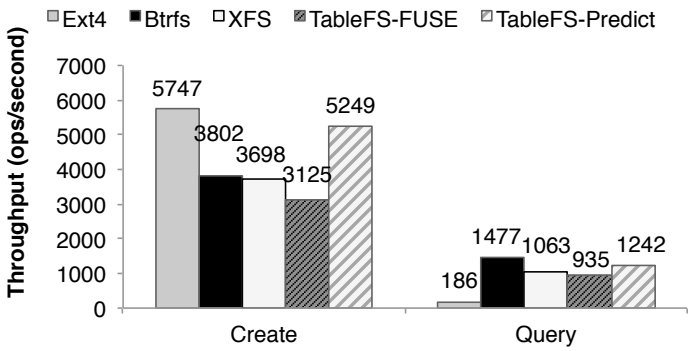
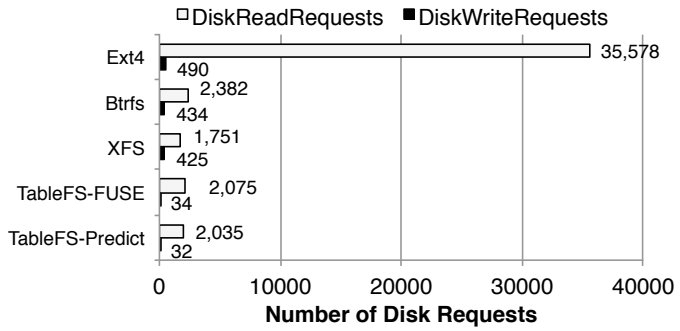
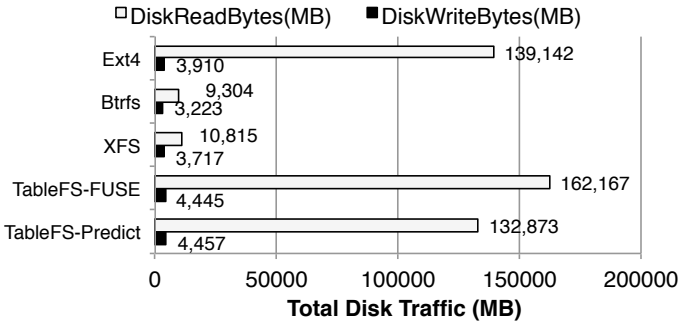


Figure 4.17: Average throughput in the create and query workloads on an Intel 520 SSD for five tested file systems.

procedure. For use with solid state disks, TableFS can be further optimized to reduce read amplification by using more advanced indexing techniques that are introduced in later chapters.



(a) Disk Requests



(b) Disk Bytes

Figure 4.18: Total number of disk requests and disk bytes moved in the query workload on an Intel 520 SSD for five tested file systems.

4.3.5 Column-Style Metadata Storage Schema

This section demonstrates the trade-offs between the two metadata storage formats used in TableFS: the two table column-style storage schema and the one table LevelDB only schema. We use a two-phase key-value workload that inserts and reads 3 million entries containing 20-byte keys and variable length values. The first phase of the workload inserts 3 million entries into an empty table in either a sequential or random key order. The second phase of the workload reads all the entries or only the first 1% of entries in a uniformly random order. To ensure that the out-of-RAM performance is tested, we limit the machine's available memory to 300MB so the entire data set does not fit in memory.

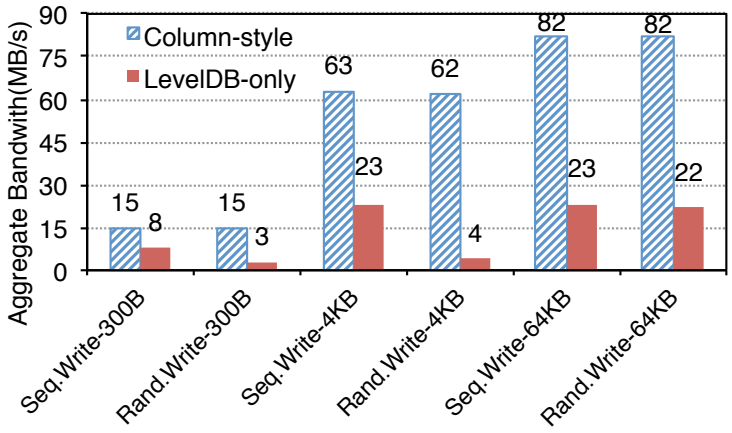


Figure 4.19: Average benchmark bandwidth when inserting 3 million entries with different sizes into the column-style storage schema and the LevelDB-only on a single server.

Insertion Throughput: The column-style schema sustains an average insert rate of 56,000 320-byte key-value pairs per second for sequential insertion order, and 52,000 pairs per second for random insertion order. Figure 4.19 shows the insertion bandwidth for different value sizes (disk is fully saturated in all cases). Column-style is about two to four times faster than LevelDB-only in all cases. Its insertion performance is insensitive to the key order because most of its work is to append key-value pairs into the log file. By only merge-sorting the much smaller index, column-style incurs fewer compactions than the LevelDB-only format, significantly reducing hidden disk traffic.

Read Throughput: Table 4.1 shows the average read throughput in the second phase of the workload (with 320-byte key-value pairs). The column-style schema is about 60% faster than LevelDB-only for random reads after sequential writes, but the former is about 10 times slower in the *read hot after random write* case. This is because the read pattern does not match the write pattern in the data files, and unlike LevelDB-only schema, column-style does not sort entries stored in data files. In this workload, LevelDB-only caches key-value pairs more effectively than column-style.

In summary, column-style has far smaller write amplification than one-table-only schema. Therefore, column-style is suitable for write critical workloads that are not read intensive or that have read patterns that match the write patterns. For

	random read after sequential write	random read after random write
Column-style	350 op/s	139 op/s
LevelDB-only	219 op/s	136 op/s
	read hot after sequential write	**read hot after random write**
Column-style	154K op/s	8K op/s
LevelDB-only	142K op/s	80K op/s

Table 4.1: *Average throughput when reading 5 million 320B entries from the column-style schema and original LevelDB-only on a single server.*

example, distributed checkpointing, snapshot and backup workloads are all suitable for column-sytle storage schema. Column-style is also suitable for many modern storage devices whose life cycles are hampered by high write amplifiaction such as solid state disks. On these devices, it is easy to compensate for read operations since solid state disks provides fast random read and using additional memory for indexing and caching can be also cost-effective.

4.4 Summary

File systems for modern storage devices have long suffered low performance when managing huge collections of small files. TableFS uses write-optimized indexes to pack small things (directory entries, inode attributes, small file data) into large on-disk files with the goal of suffering fewer seeks when seeks are unavoidable. The TableFS implementation, even hampered by FUSE overhead, LevelDB code overhead and pessimistically padded inode attributes, still performs as much as 10 times better than state-of-the-art local file systems in extensive metadata update workloads in hard disks.

This chapter mainly discusses the modular design of TableFS and schema optimization for metadata workload with different I/O characteristics. The evaluation reveals the impact of compactions on write amplification. Reducing write amplification is important for storage devices such as solid state disks and shingled disks, and is more difficult than optimizing read operations that can be compensated by using more memory. Later chapters will discuss the trade-off among read amplifica-

tion, write amplification and memory usages in external memory indexing, and the systematic approach to balance the three for file system workloads.

Chapter 5

SlimFS: Space Efficient Indexing and Balanced Read-Write Performance

TableFS has demonstrated that key-value stores can be used as a backbone to scale file system metadata management on today's storage devices. In TableFS, we used the out-of-box LSM-tree implementation LevelDB to represent the file system metadata, which out-performs modern Linux local file systems by as much as an order of magnitude. The rationale of using LSM-tree is that LSM-tree is highly write optimized and has lower write amplification compared to traditional B-tree. This optimization brings huge improvements, especially for modern storage devices such as solid-state disks and shingled disks. Through the evaluation section of the previous chapter, we learned that the on-disk layout used in LSM-tree actually trades read performance for better write performance. To achieve comparable performance, LSM-tree uses filters and in-memory indexes to speed up the lookup operation. We also identify one key limitation in using LSM-tree: the compaction procedure still generates considerable write amplifications. One natural question to ask is whether it is possible to further reduce write amplification while using in-memory filter and indexes to maintain fast read performance.

The goal of this chapter is to explore techniques that can achieve better balance among three factors: read amplification, write amplification and the use of memory resource for file system metadata management and key-value storage systems. One important insight we have discovered is that file system semantics can make critical difference in the system design. For example, for range queries like `readdir`, the POSIX standard does not require the list returned to be sorted. As in TableFS, the primary key is divided into two fragments: a prefix x (parent directory's inode

number) and a suffix y (filename). `readdir` only needs to iterate through all the keys that share the same prefix x, without any ordering requirement on y. We define this kind of ordering requirement as **Semi-Sorted**. Semi-sorted order is stronger than hash order. We will show that file objects can be indexed in semi-sorted order by using highly-compressed hash-based key schema [LFAK11].

Additionally, most metadata write operations fall into one category called **Non-Blind Writes**: these writes first perform a read operation on the same key before inserting or updating a key. For example, file creation requires checking for the file's existence before the actual file creation, and update operations such as `chmod` and `utime` are inherently all read-modify-write operations. By exploiting these properties, we will show that it takes only a little extra memory resource and no additional storage access overhead to maintain richer indexes that have faster in-memory lookup performance and better tail lookup latency for key-value stores. (Here latency means the number of disk reads used by each lookup operation since it is the biggest source of latency in the key-value store). This also enables us to use alternative on-disk layout for LSM-tree to further reduce its write amplification while maintaining read performance.

This chapter demonstrates that file system characteristics can be leveraged to design a more efficient LSM-tree implementation optimized for file system metadata processing and solid-state disks. The new file system prototype called SlimFS made the following improvements over the original LevelDB: 1) a redesign of LevelDB's SSTable index with space-efficient data structures specialized for file system workloads; 2) a novel membership filter that bounds the number of disk reads in a multi-level log-structured key-value store under the worst case; 3) an analytical model automatically choosing different types of in-memory indices for each level in LSM tree to achieve optimal read and write amplification based on workloads. We have conducted a thorough evaluation of our file system prototype implementation based on a modified version of LevelDB called SlimDB. Our experiments show that by applying these design techniques, SlimFS can be three times faster for file creates, two times faster for file stats, using less memory to cache metadata indices, and exhibiting better tail latency in read operations, relative to the original TableFS using a general-purpose LSM-tree implementation such as the original LevelDB, RocksDB, or HyperLevelDB.

5.1 The Analysis of Log-Structured Designs

In previous chapter, we have shown how LSM-tree out-performs traditional B-tree in terms of write amplification. This section discusses another log-structured design called "Stepped-Merge" that has different read and write amplification compared to the original LSM-tree. Stepped-Merge algorithm creates sub-levels within each level that may have overlapping key ranges among sub-levels. Its on-disk layout avoids unnecessary compaction overhead, while increasing read latency. By comparing these two different log-structure designs, we will demonstrate that the read-write trade-off is inherent in the on-disk indexing data structure, and the utilization of compact in-memory indexing is the right direction to balance the read/write performance.

5.1.1 I/O Cost Analysis of Log-Structured Merge Tree

As introduced in Chapter 4, an LSM-tree (LevelDB) builds a multi-level tree-like structure to progressively sort key value entries. The size of each level follows an exponential growth pattern such that the size of a level is r times larger than the previous level size. With this exponential pattern, there are at most $log_r N$ levels, where N is the total number of unique keys (is approximately equal to the size of the last level). Thus the worst-case lookup incurs $O(log_r N)$ random reads to the disk by accessing all levels.

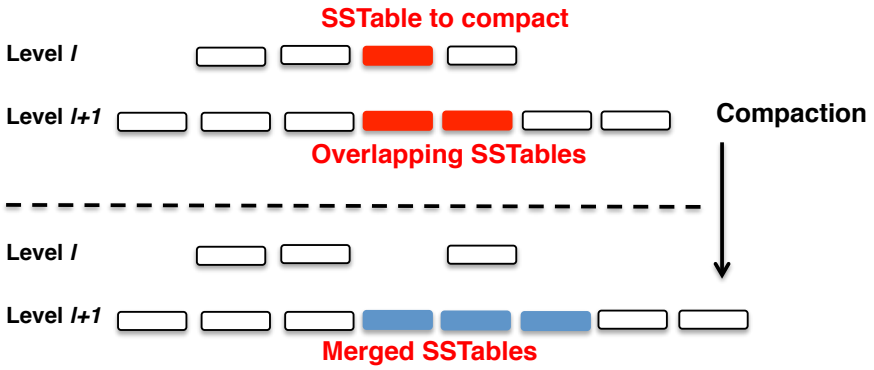


Figure 5.1: Illustration of the compaction of LSM-tree.

The use of buffering and exponential growth pattern in an LSM-tree leads to a write amplification different from a B-Tree. Our definition of write amplification means the expected amount of data actually written to the secondary storage divided

by the data size of the inserted entry. It measures the I/O overhead of each insertion operation. As shown in Figure 5.1, background compaction will move all SSTables from one level k (k = 0,1,2,...) to its next level $k+1$ by merge-sorting the SSTables of two levels. In the worst case, the key range of level k overlaps the entire key range of level $k+1$, which requires merge-sorting all the SSTables in the both levels. Therefore, for an entry in Level $i+1$, it may get involved in $r/2$ times compaction with Level i on average before it gets compacted into Level $i+2$. This means that the write amplification of moving data from one level to its next level is r in the worst case (including both I/O reads and writes during the compaction). Assuming one entry reaches level $k+1$, the write amplification for inserting this entry goes up to $k \times r$. The write amplification per insertion is then $O(rlog_rN)$ for the majority keys stored in the bottom level. Because entries are transferred in batches during compaction, the amortized I/O cost per insertion is $O(\frac{1}{B}rlog_rN)$ in the worst case where B is the number of entries in a write batch. The total number of levels in an LSM-tree can easily reach to 4 or 5, and the common values of r are between 8 and 16. The average write amplification for inserting an entry can be as large as 80 when LSM-tree stores lots of entries. Such high write amplification can consume most of the I/O bandwidth and wear out solid state disks quickly.

5.1.2 Stepped-Merge Algorithm: Reducing Write Amplification in Compaction

Stepped-Merge uses a different organization and compaction strategy to manage SSTables [JNS+97]. The main purpose of compaction is to make room for newly inserted entries by integrating SSTables from Level i to Level $i+1$. The major source of write amplification comes from the fact that the compaction procedure has to merge-sort one SSTable with all of the overlapping SSTables in the next level, which amplifies the compaction overhead.

Based on this observation, Stepped-Merge uses a different organizational layout and compaction strategy to manage SSTables [JNS+97]. As shown in Figure 5.2, Stepped-Merge divides the SSTables in each level into r sub-levels. The size limit of each level is still the same as the LSM-tree. However, when compacting SSTables in Level i, Stepped-Merge does not merge-sort SSTables in Level i with tables in Level $i+1$ as the LSM-tree does. Instead, all sub-levels in Level i are r-way merge-sorted and inserted into Level $i+1$ as a new sub-level. The total amount of transferred data during merge-sorting r sub-levels is roughly the same as the total amount of data stored in these sub-levels. By doing so, the amortized cost of migrating an

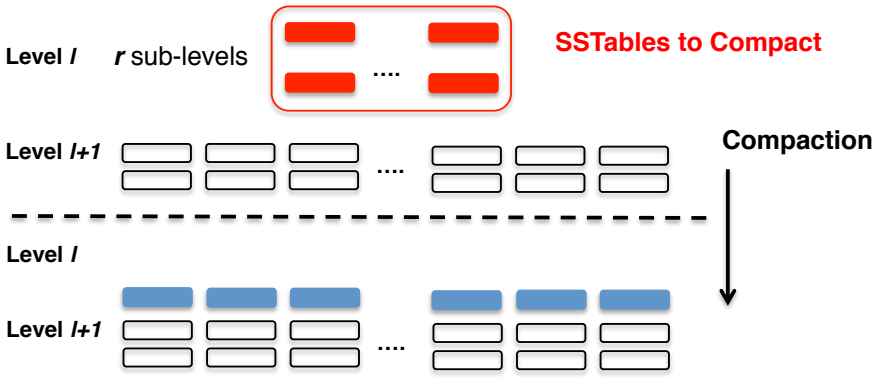


Figure 5.2: Illustration of Stepped-Merge algorithm.

entry from Level i to Level $i+1$ is reduced to only two times its size. Since each long-lived, inserted entry is written at each level only once, the write amplification for an entry to reach level $i+1$ is i. Thus, the amortized I/O cost of an insertion in Stepped-Merge decreases to $O(\frac{1}{B}log_r N)$. On the other hand, a lookup operation in Stepped-Merge has to check $rlog_r N$ sub-levels to locate a key, which costs $O(rlog_r N)$ random reads from disk in the worst case.

5.1.3 Optimizing In-memory Indexes and Filters

Although the stepped-merge algorithm can reduce the write amplification, its read performance degrades because the algorithm has multiple overlapping sub-levels within each level for a lookup operation to read in order to find a particular entry. To avoid high read latency while maintaining low write amplification, one potential solution is to increase the effectiveness of a store's in-memory indexes and filters. These enhanced in-memory data structures can better pinpoint where entries might and will not be, and therefore can avoid unnecessary disk accesses.

Figure 5.3 shows the main components of a typical LevelDB SSTable, which is the format used by LevelDB to store data. Each SSTable stores its data in sorted order across an array of data blocks. The size of each data block is configurable and is usually 4KB. In addition to these data blocks, a special index block is created that maps each data block to its key range. This index block consists of all the largest keys of every data block. Along with this index block, a Bloom filter is also used to record the existence of all the keys in the table [Blo70]. For each lookup operation, LevelDB first checks the Bloom filter to ascertain the non-existence of a key, else

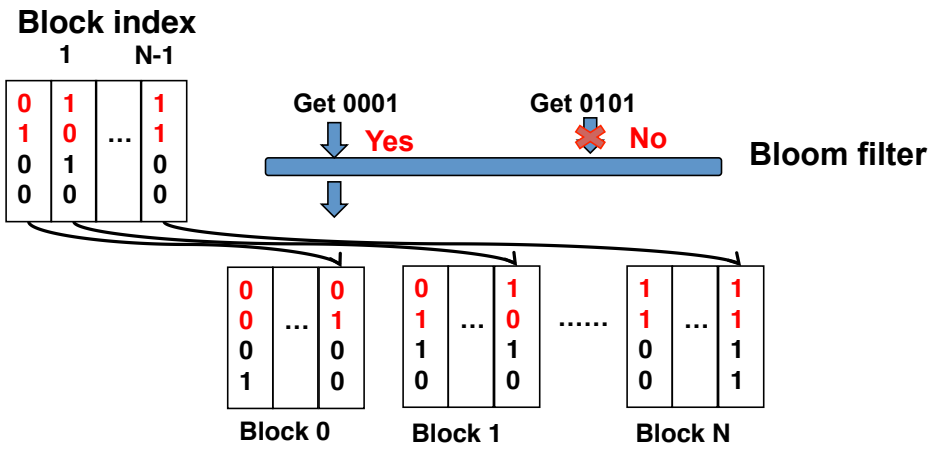


Figure 5.3: Illustration of the basic index format of LevelDB's SSTable and its read path. The keys follow a semi-sorted order, so each key has two parts: the prefix (red) and the suffix (black).

it uses the index block to find the right data block. In Figure 5.3, for example, to lookup key "0001" the lookup process will first go through the Bloom filter and will find that the key may be in the SSTable. It then checks the corresponding index block, which will lead the lookup to data block 0. To lookup key "0101", the lookup process will be stopped by the Bloom filter as key "0101" was never inserted into this example table.

Since the Bloom filter is a probabilistic data structure with a false positive rate, a lookup operation may fetch a data block that does not contain the target key, thus adding additional read latency. On the other hand, all SSTable indexes and filters in many key-value stores are often stored in memory in a LRU cache with a fixed memory limit. Making high quality indexes and filters more compact will allow more entries can be precisely indexed and avoid loading block indexes and filters from the disk. Because the quality and the size of block indexes and filters are key to ensuring good read performance, we will show in the following sections how to improve indexes and filters by leveraging common key-value workload characteristics.

In the next section, we show how to make in-memory indexes and filters more compact such that more indexes and filters can be cached in the memory. In this paper, we target three strategies for improvement. First, we aim to make in-memory indexes and filters more compact such that more indexes and filters can be cached in the memory. Section 5.3.1 shows that the original SSTable block index can be

compressed further by using an advanced compact data structure and special key schema that is optimized for semi-sorted data. Second, we aim to improve the tail latency of read operations even for the data layout used by a stepped-merge algorithm. In Section 5.3.2, we show that this can be achieved by using an augmented filter design that has similar memory cost as the Bloom filter but more powerful semantics and a lower false positive rate. Third, we observe that for a multi-level log-structured store, each level may use a different combination of data layout design, index, and filter data structure. Section 5.4 will introduce an analytic model that finds the optimal combination for each level under different workloads and memory constraints.

5.2 The Design Overview of SlimFS

5.2.1 The SlimFS Architecture

SlimFS is designed as middleware layered on top of FUSE in the same way as TableFS does. SlimFS stores file system metadata into a key-value database. Similar to TableFS, the file system design is modularized such that the underlying database can be any key-value store that supports `PUT`, `GET`, `DEL` and `SCAN`. The only difference is that SlimFS uses a slightly different key schema for metadata store as listed in Table 5.1. SlimFS's metadata store aggregates directory entries and inode attributes into one store. To link together the hierarchical structure of the user's namespace, the rows of the table are ordered lexicographically by the composite key consisting of 64-bit hash values of inode number of an entry's parent directory and its entry name (final component of its pathname). Thus, all entries within the same directory are stored consecutively. The use of hash function for both inode number of parent directory and filename is for better index compression that will be explained in the next section.

metadata	$\{h(\text{parent-inode\#}), h(\text{name})\} \rightarrow \text{inode}$

Table 5.1: Key-value store schema used in SlimFS.

SlimDB is the enhanced log-structure design used by SlimFS, which combines a compact index and filter with the stepped-merge algorithm to achieve better performance both reads and writes. It also uses a multi-store design such that the

data layout and index used by each level can be differently tuned to meet workload requirements.

5.2.2 SlimDB's Compact Index and Multi-Store Design

The goal of SlimDB is to achieve low read and write amplification. The following summarizes the characteristics of the indexes and filters used in SlimDB:

- *Three-level Block Index*: Our three-level block index replaces the original block index used in LevelDB's SSTable. This new index is specially optimized for semi-sorted data. It features a memory cost that is as small as 0.7 bits per key.
- *Multi-level Cuckoo Filter*: The multi-level cuckoo filter is a replacement of Bloom filters for the stepped-merge algorithm. When searching for a key using a multi-level cuckoo filter, the filter returns the most recent sub-level containing the target key if the key appears to exist. Similar to Bloom filters, the multi-level cuckoo filter is a probabilistic data structure which may give the wrong answer if the key does not exist. But even in the worst case, the lookup procedure will only need to access a SSTable in one sub-level in a workload with only blind writes.

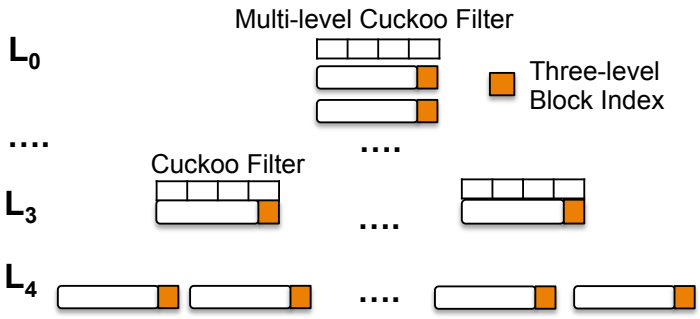


Figure 5.4: The use of multi-store design in SlimDB. Filters and indexes are generally in-memory, and for a large store SSTables are mostly on disk.

Combining different data layouts and indexes gives rise to key-value stores with different read, write amplification, and memory costs. For example, we can combine a multi-level cuckoo filter with a stepped-merge algorithm. Together they can have

lower write amplification than an original LSM-tree but may require more memory resources. There is no one combination that is strictly better than all other combinations. However, the multi-level structure used by many log-structured store designs allows for a flexible use of different key-value store combinations at each level [LFAK11]. As we shall show, these multi-level stores are able to leverage a mix of key-value store designs to balance read amplification, write amplification, and memory usage. Figure 5.4 gives an example of the multi-store design in SlimDB. Level 0, 1 and 2 all use the data layout of the stepped-merge algorithm, with multi-level cuckoo filters and three-level block indexes. All filters and indexes are cached in memory. But Level 3 and Level 4 use the data layout of the original LSM-tree, and cache three-level block indexes in memory. Further, Level 3 but not Level 4 caches Bloom filters in memory.

The following sections explains our novel indexes and filters. Section 5.4 will show how to use our proposed analytic model to automatically select basic key-value store designs for each level to meet resource and performance constraints.

5.3 Design of Compact Index and Filter in SlimDB

5.3.1 Three-Level Index: Compact Block Index for SSTable

In LevelDB's original SSTable format, key-value pairs are sorted and packed into data blocks. As shown in Figure 5.3, each SSTable file contains an index block at the end of the file that stores the full key of each data block's last entry. Without caching the block index, reading an entry from an SSTable requires two block reads: one to load the index block and the other to read the actual entry. Since the size of an SSTable data block is usually set to 4KB and the typical size of an entry in many applications (e.g, file system metadata, feature storage in recommendation system) might be smaller than 256 bytes [LFAK11], each block stores, say, at most 16 entries. As LevelDB's block index stores a full key (e.g. 16B) for each data block, the average space required to store a key might be 16B / 16 = 8 bits. The block index representation can be plug replaced without impacting the general LSM organization and execution. Our goal is to to employ sophisticated compression schemes on the index block to trade more CPU cycles for fewer storage accesses through higher cache hit rates when fetching a random entry from an on-disk SSTable.

Different from LevelDB, which is designed for totally ordered keys, SlimDB only needs to support semi-sorted keys that consist of a prefix and a suffix. This means

that the keys in a SlimDB SSTable only need to be sorted by their prefixes. This enables us to use entropy-coded tries (ECT) [LFAK11] to compress prefixes and suffixes separately in the index block. ECT can efficiently index a sorted list of fixed-sized hash keys using only 2.5 bits per entry on average. In this section, we construct a semi-sorted block index with ECT to use only 1.9 bits to index an entry to its block, which is 4X smaller than the LevelDB method.

Entropy-Encoded Trie Basics: Given an array of n distinct keys that are sorted by their hash order, an ECT data structure is able to map each input key to its rank ($\in [0, n-1]$) in the array. As shown in Figure 5.5, each ECT is a radix tree that stores a set of keys where each leaf node represents one key in the set and each internal node denotes the longest common prefix shared by the keys under the subtree rooted by this internal node. For each key stored, ECT only preserves the shortest partial key prefix that is sufficient to differentiate it from other keys. Although ECT can index a set of keys, it cannot check key membership, so additional data structures (such as bloom filters) are still needed to avoid false lookups.

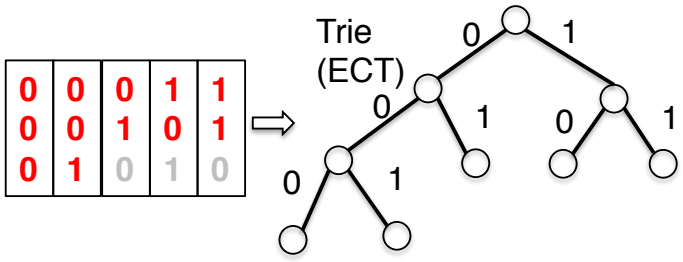


Figure 5.5: ECT transforms a list of sorted key hashes into a radix tree that only keeps the shortest prefix of each key that is enough to distinguish it from other keys.

Because all keys are hashed, ensuring a uniform distribution, a combination of Huffman coding and Elias-gamma coding is able to greatly compress the keys in each trie. Details of how to compress the trie are described in [LFAK11].

Three-Level Index Design: Unlike hash tables, SlimDB is designed to retain the semi-ordering of keys. Using ECT alone is not sufficient to serve as a block index for SlimDB. In SlimDB both key fragments are hashed, so it is possible to use ECT to index each key fragment individually, leading to a two-step search procedure: first find a group of SSTable blocks that contain all keys that share the same prefix as the sought key; then locate the specific SSTable block containing the sought key from the group of blocks returned by the first step.

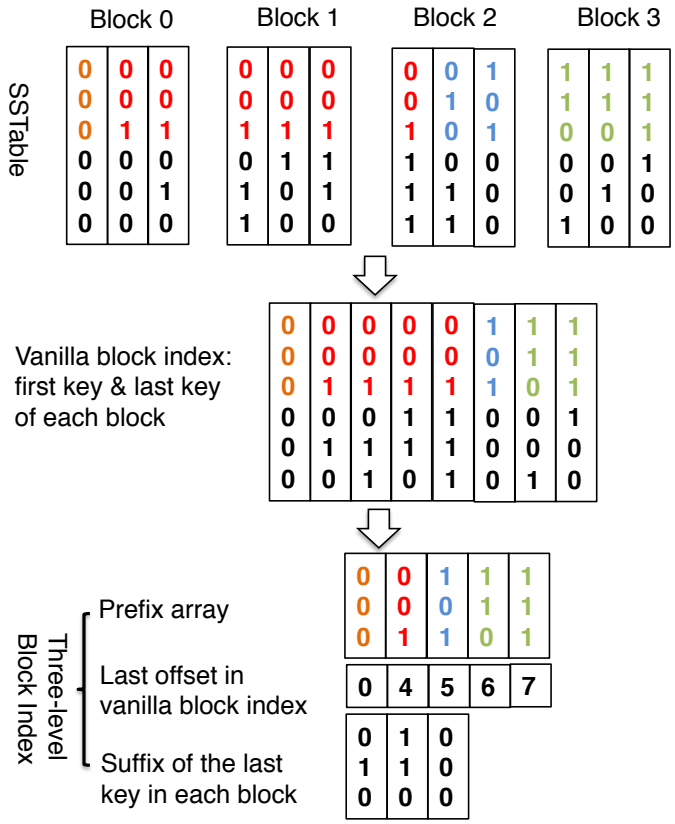


Figure 5.6: An example three-level block index for a SlimDB SSTable.

As shown in the example in Figure 5.6, to search a key in an array of blocks, the vanilla approach is to create a block index consisting of the first and the last key in each block. The construction of three-level index is based on compressing this block index. The procedure of consturcting the three-level index is shown in Algorithm 1. First, the prefix of these block keys are stored separately in a prefix array, where only one prefix is preserved and duplicated prefixes are removed. We use ECT to compress this prefix array, which constitutes the first level of our index. This level allows us to map each key, using its prefix, to its rank in the prefix array, and this rank in turn becomes the input to the second level of our three-level index, which we now describe.

The second level of our three-level index takes the rank in the prefix array and maps it to one or more SSTable data blocks that contain entries matching this specific prefix key. In this integer array each element stores the offset in the vanilla block

index of the last block key containing the corresponding prefix in the prefix array. For example, the last block containing "001" is Block 2, and its offset in the vallina block index is 4. Therefore its corresponding element in the second level is 4.

Algorithm 1: construct(b)

```
Data: b: vanilla block index;
Result: pa: prefix array; ca counter array; sa: suffix array;
n = 0
for each key k in b do
    if k.prefix != the last prefix in pa then
        pa[n] = k.prefix
        ca[n] = 1
        n = n + 1
    end
    else
        ca[n] = ca[n] + 1
    end
end
for each block i except the last block do
    k1 = b[i * 2 + 1]
    k2 = b[i * 2 + 2]
    if k1.prefix != k2.prefix then
        sa[i] = empty
    end
    else
        sa[i] = shortest common prefix that distinguishes k1.suffix and k2.suffix
    end
end
pa = compress pa using ECT encoding
ca = compress ca as rank/select dictionary
return (pa, ca, sa)
```

Through the mappings defined by the first two levels, a lookup procedure is able to retrieve a list of potential SSTable blocks that contain the sought key's prefix. To finally locate the SSTable block whose range covers the sought key, the last step is to binary search through all potential SSTable blocks using the suffix of the last entry from each block. Similar to the first level index, the array of suffixes of block keys sharing the same prefix can be compressed by using ECT.

To optionally speed up the lookup process without using ECT, our three-level index can store an array of partial suffixes instead: each partial suffix is the shortest unique prefix of the original suffix that distinguishes a pair of suffixes from the two adjacent SSTable blocks. For example, when searching for the key "001000", we find it must reside between Block 0 to Block 2 inclusive, based on the first two level indexes, as shown in Figure 5.6. To locate its block, we use its suffix "000" to complete a binary search among the array of suffixes ("010", "110") that differentiate the three candidate block groups. Since "000" is smaller than "010", "001000" can only be stored in Block 0. The lookup procedure is shown in Algorithm 2.

Algorithm 2: search(pa, ca, sa, x)

Data: pa: prefix array; ca counter array; sa: suffix array;
Result: loc: block offset;
i = get rank x.prefix from pa using ECT lookup procedure
lb = select$(i-1)$ from ca, which gets the first block having prefix $pa[i]$
rb = select(i) from ca, which gets the last block of prefix $pa[i]$
loc = binary search x.suffix among $sa[lb..rb]$
return loc

Analysis of Three-Level SSTable Index: For the first level, the prefix array needs to store at most two prefixes per SSTable block, and all prefixes are hash sorted which can be used for ECT encoding. Since ECT costs 2.5 bits per prefix key on average, the first level costs no more than $2 \times 2.5 = 5$ bits per SSTable block.

For the second-level index that records, the last block's offset per prefix, we can represent it with a rank/select dictionry[Jac88]. It first uses delta encoding to calculate the difference between two offsets. Because the sum of these deltas cannot exceed the number of blocks in the SSTable, we can then use unary coding to represent the delta as a bit vector, with no more than two bits per block in an SSTable. Optionally, to speed up searching in this array, a sum and pointer enables quick skipping one set of k deltas, this can be added to the bit vector for every k deltas. If the size of the sum and a pointer is 16 bits and $k = 32$, then building this array costs $2 + 16/k = 2.5$ bits per group.

The third-level index that records per-block last suffixes costs 2.5 bits per block on average if using ECT. If using an array of partial suffixes instead of ECT, Monte carlo simulation of all possible arrays of partial keys shows that the average length of the partial key that separates two adjacent suffixes is about 16 bits. Another 6

bits is used to record the length of each partial key. So the average cost of the faster lookup third-level index is 22 bits per block (using the array of partial keys).

Summing the average-case cost of all three index levels, the three-level SSTable consumes 10 $(5 + 2.5 + 2.5)$ bits per SSTable block using ECT on the third-level index. Using 16 key-value items per block, memory overhead is $10/16 = 0.7$ bits per key, much smaller than LevelDB's 8 bits per key. If using the array of partial keys for faster lookup in third-level index, the memory overhead is $(5+2.5+22)/16 = 1.9$ bits per key (still 4X better than LevelDB).

5.3.2 Multi-Level Cuckoo Filter: Improve Tail Latency

In-memory filters are data structures commonly used by many high performance key-value stores to efficiently test whether a given key is not found in the store before accessing the disk. Most of these filters are probablistic data structures that perform false positive accesses. One main source of long tail latency in the read path of a stepped-merge store lies in false positive answers given by Bloom filters at multiple levels when looking for a key. We propose a new filter design, called a multi-level cuckoo filter, that can limit the number of disk reads in such cases. This new filter design uses the cuckoo filter as a building block. Cuckoo filters are similar to Bloom filters but have properties like lower memory cost and fingerprint-based filtering [FAKM14]. This section introduces how the design of our multi-level cuckoo filter improves the read tail latency of key-value stores by leveraging these properties.

Cuckoo Filter Basics: A cuckoo filter extends standard cuckoo hash tables [PR04] to provide membership information. As shown in Figure 5.7, a cuckoo hash table is a linear array of key buckets where each key has two candidate buckets calculated by two independent hash functions. When looking up a key, the procedure checks both candidate buckets to see if the entry exists. An entry can be inserted into any one of two candidate buckets that is vacant. If both are full, then the procedure displaces one existing entry in either bucket and re-inserts the victim to its alternative bucket. The displacement procedure repeats until a vacant bucket is found or the maximum number of displacements is reached (e.g, hundreds of tries). In the latter case, the hash table is declared to be full, and an expansion process is executed. Although cuckoo hashing may execute a series of displacements, the amortized I/O cost of insertion operation is $O(1)$.

Cuckoo hashing can achieve higher space occupancy by using more hash functions as well as extending the buckets to have more than one slot to allow several entries

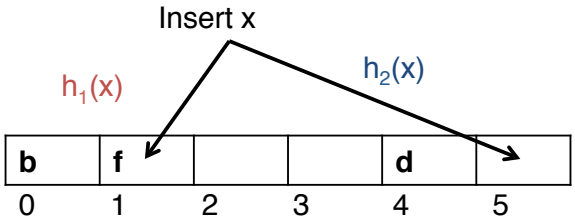


Figure 5.7: Illustration of Cuckoo Hashing.

to co-exist. Nikolaos et. al. present an analysis of the maximum possible occupancy ratio, showing that with 2 hash functions and a bucket of size 4, the table space can be 95% filled [FKP11].

Cuckoo hashing can be used directly to implement a membership query. But since the hash table stores the full key, it has high space overhead compared to a Bloom filter. To save space, a cuckoo filter [FAKM14] only stores a constant-sized hash fingerprint of any inserted entry instead of its original full key. This results in changes to the insertion procedure. Storing only fingerprints in the hash table prevents inserting entries using the standard cuckoo hashing approach, since it prevents the algorithm from calculating the entry's alternative position. To overcome this limitation, the cuckoo filter uses the fingerprint to calculate an entry's alternative bucket rather than the key itself. For example, the cuckoo hash indexes of the two candidate buckets of an entry x ($h_1(x)$ and $h_2(x)$) are calculated as follows:

$$h_1(x) = hash(x),$$

$$h_2(x) = h_1(x) \oplus hash(x\text{'s fingerprint})$$

Obviously, the two functions are symetric since $h_1(x) = h_2(x) \oplus hash(x\text{'s fingerprint})$. This design causes the two hash functions to be less independent of each other, therefore the collision rate in the hash table is higher than that of the standard cuckoo hashing table. However, by selecting an approriate fingerprint size f and bucket size b, it can be shown that the cuckoo filter is more space-efficient than the Bloom filter when the target false postive rate is smaller than 3% [FAKM14].

Integration with Multi-level Stores: Figure 5.8 depicts the integration of a multi-level cuckoo filter with multi-level key-value stores. The multi-level cuckoo filter has two separate tables: the primary table and the secondary table. The primary table is a cuckoo filter variant that stores both the fingerprint, $f(x)$, and the level number of each entry, $l(x)$. Different from the basic cuckoo filter, the

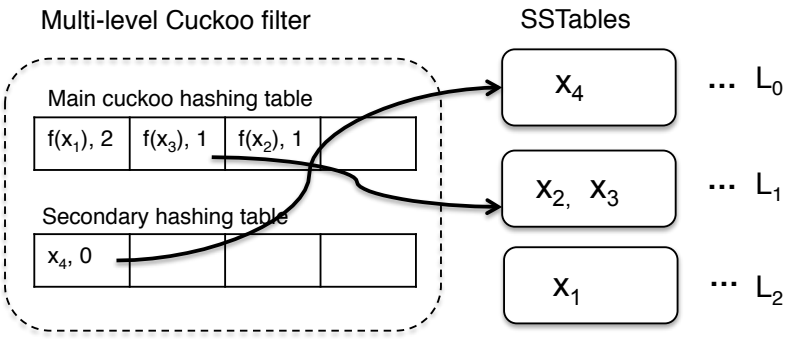


Figure 5.8: Illustration of integrating cuckoo filters with multi-level indexes and a secondary table. If a key has a hash collision with some key in the primary hashing table, the key will be put into the secondary hashing table. Each lookup first checks in the secondary table and then the primary table.

level number stored in the primary table can be used to locate the sub-level in the LSM-tree in which the target entry is actually present.

The secondary table is used to bound tail latency by storing special entries with their full keys. The reason for having the secondary table is to cope with the case that multiple entries may have the same fingerprint. In such cases, the primary table would need to keep multiple copies of the same fingerprint with all the associated level numbers. To locate the level that actually contains the sought entry, it would be necessary to perform a disk read for each level associated with the conflicting fingerprint in the worst case. A straightfoward method to reduce the number of disk reads in the worst case is to avoid fingerprint conflicts in the primary table, which means that each fingerprint must be unique and can only be associated with one level number in the primary table. To maintain this property, the multi-level cuckoo filter uses the secondary table to record the full key for each entry, after the first, having a conflicting fingerprint in the primary table.

With a secondary table for conflicting entries, the procedure of looking up an entry in multi-level cuckoo filter is still straightforward, as shown in Algorithm 3. When searching for an entry, first search the secondary table to see if there is any matching full key. If found, the level number of this entry can be retrieved from the secondary table; otherwise, continue to check the primary table of the multi-level cuckoo filter. With the secondary table in memory, the worst case lookup performs at most one disk read since it is guaranteed that there is only one copy of each fingerprint in the primary table.

To maintain uniqueness of fingerprints in the primary table, the insertion procedure in multi-level cuckoo filters must follow certain rules as shown in Algorithm 4. The multi-level cuckoo filter is built along with the multi-level stores, meaning that newer entries are always inserted into newer (lower) levels. When inserting an entry, if its fingerprint already exists, then we check whether the fingerprint in the primary table is derived from the same key. If it is derived from the same key, then the entry must have a newer level number, and therefore we only update the level number associated with the key in the primary table. Otherwise, the entry is put into the secondary table due to the conflict of the fingerprint. If the fingerprint does not exist, then the entry can be safely inserted into the primary table. For example, as shown in Figure 5.8, when inserting x_4, it might happen that x_4's fingerprint is the same as x_1's. Thus, x_4 has to be put into the secondary table. However, if we insert x_1 with level 0 that is newer than level 2, then only its level number in the primary table needs to be updated.

Algorithm 3: lookup(key x)

```
Data: c: primary table; t: secondary table
l = t.lookup(x)
if l is not NULL then
    return l
else
    f = fingerprint(x)
    return c.lookup(f)
end
```

Algorithm 4: insert(key x, level l)

```
Data: c: primary table; t: secondary table; store: on-disk store
f = fingerprint(x)
l' = c.lookup(f)
if l' is not NULL then
    if store has no x in level l' then
        t.insert(x, l)
        return
    end
end
c.insert(f, l)
```

To verify whether the conflicting fingerprint comes from the same key in the primary table, it is not necessary to perform disk reads to retrieve the full key if writes are non-blind. Our strategy then is to take advantage of non-blind writes to avoid unnecessary disk traffic when possible. For example, in file system metadata workloads under the POSIX standard, all metadata write operations are non-blind, which means that a read operation will have always been performed on a key before any write of that key. The existence check operation done by prior read avoids additional disk reads needed for the multi-level cuckoo filter to verify whether the same key exists in other levels.

If blind writes do happen in the workload, the same key can be inserted into both the primary and secondary table if the insertion procedure does not read the full key from the disk. In this case, however, the number of keys stored in the secondary table will exceed reasonable space targets, so our algorithm will stop inserting new entries into the secondary table, and the multi-level cuckoo filter will exhibit similar false positives and tail latency distribution as the original Bloom filters.

Memory Footprint Analysis: While it might seem that the multi-level cuckoo filter may use $log_2(L), L \approx \log(N)$ more bits per entry compared to the traditional Bloom filters used in LevelDB, the primary table of the multi-level cuckoo filter actually has the same memory cost even in the worst case. To see why, assume the desired false positive rate for the multi-level cuckoo filter is ϵ. For traditional methods [Lev11] that use a Bloom filter for each SSTable, the overall false positive rate is at least $1-(1-\alpha)^L \approx L \cdot \alpha$ if the false positive rate in each level is α and they are independently distributed. In order to achieve the same false positive rate as the multi-level cuckoo filter, a Bloom filter's α must be ϵ/L. The space requirement for any Bloom filter to achieve a false positive rate α is at least $\log_2 1/\alpha$. So the overall cost of the traditional method is $log_2(1/\alpha) = log_2(1/\epsilon) + log_2(L)$ per entry, which is the same as the multi-level cuckoo filter.

The average size of the secondary table is proportional to the false positive rate of the primary table. To see why, assume that there are n elements that need to be inserted into the primary filter, and the primary filter has a false positive rate ϵ. The expected number of entries stored in the secondary table is the expected number of entries that generate false positive anwers, which is $n \times \epsilon$.

SlimDB uses a multi-level cuckoo filter with a less than 0.1% false positive rate. The size of each item stored in the secondary table is the sum of the length of the full key and the size of the level number which is $128 + 8 = 136$ bits. The secondary table increases the memory overhead by $136 \times 0.1\% = 0.136$ bit per entry, which is $0.136/16 = 0.8\%$ of the original Bloom filter's cost.

5.3.3 Implementation of SlimDB

The implementation of SlimDB is based on RocksDB [Hyp13a], and has about 5000 lines of code changes. RocksDB has a modular architecture where each component of the system exports the same, basic key-value interface including the memory-index and on-disk SSTable. This allows us to easily add new filter policy and block index into its SSTables. In SlimDB, items are still sorted according to their hashed prefix and suffix. Thus, point queries do not require additonal changes to RocksDB other than filters and block indexes. For prefix scan (e.g. list all entries sharing the same prefix) with stepped-merge algorithms, its procedure has to maintain an SSTable iterator in each sub-level, which is slower than traditional LevelDB. Since items are sorted by hashed prefix, SlimDB cannot support scan across prefixes with one index. To support fully ordered key scan in some workloads, SlimDB needs to maintain another secondary index that stores all the prefixes without hashing.

The use of stepped-merge algorithm in SlimDB is similar to the procedure described in LSM-trie [WXSJ15]. In each sub-level, semi-sorted items are grouped into SSTables based on its hash-key range as well as the size limit of each SSTable. During each compaction, the procedure will pick all SSTables within a hash-key range from all sub-levels to do merge-sorting and put newly merged SSTables into the next level.

5.4 Analytic Model for Selecting Indexes and Filters

The level structure of an LSM-tree allows for flexible use of different storage layouts and in-memory indexes on a per-level basis. This section presents an analytic model that selects storage layout and in-memory indexes to achieve low memory usage while maintaining target performance.

The key idea of the analytic model is to account for the hardware resources utilized by the index structure in each level, including the memory cost and the I/O cost of all types of requests (such as positive reads, negative reads, and insertions). To unify I/O costs of different types of requests, we use the time spent on a random 4KB disk read as the basic measurement unit. For most storage devices, the cost of writing a 4KB block sequentially compared to random block read (denoted as w) is actually quite small. For example, the solid state disk used in our evaluation performs 4000 4KB random reads per second, and delivers 107 MB/sec sequential writes. The I/O time to write a 4KB block sequentially is $4/107/1024 \approx 0.0000365$ seconds. The I/O

time of reading a 4KB block randomly is roughly $1/4000 \approx 0.00025$ seconds. In such cases, w equals 0.146. If a 4KB block can store B entries, then the cost of inserting an entry is w/B because insertion and compaction in the log-structure design only require sequential writes.

#	Level structure	Mem. C_M	Pos. C_{PR}	Neg. C_{NR}	Writes C_W
0	LSM-Tree	0b	2	2	$\frac{rw}{B}$
1	LSM+CF	13b	2	$2f$	$\frac{rw}{B}$
2	LSM+CF+TL	15b	1	f	$\frac{rw}{B}$
3	LSM+TL	2b	1	1	$\frac{rw}{B}$
4	Stepped-Merge	0b	$r+1$	$2r$	$\frac{w}{B}$
5	SM+MLCF	15b	2	$2f$	$\frac{w}{B}$
6	SM+MLCF+TL	17b	1	f	$\frac{w}{B}$
7	SM+TL	2b	$\frac{r+1}{2}$	r	$\frac{w}{B}$

Table 5.2: The space and disk access cost of using three types of indexes. CF means cuckoo filter. TL means three-level SSTable index. MLCF means multi-level cuckoo filter.

Table 5.2 summarizes the costs of using different combinations of indexes and data layout on a per-level basis. For simplicity, our model assumes that all the SSTables within a level use the same types of index and filter, and that key queries follow a uniform distribution. For each level, if that level follows the design of an LSM-tree and has only one sub-level, then it is labeled as an LSM-tree style data layout. Otherwise, for any level having multiple sub-levels, it is labeled as a Stepped-Merge (SM) style data layout. Without any in-memory indexes, the costs of the two styles are calculated as in Section 5.1. When equipped with only a cuckoo filter or a multi-level cuckoo filter, the cost of a negative read (a read that does not find the sought key) is $2f$, where f is the false-positive rate of the filter. By caching a three-level SSTable index additionally, the average cost of retrieving an entry is reduced to f.

Once there is a cost model, then the index selection problem becomes an optimization problem whose goal is to minimize the I/O cost under memory constraints. Assume there are $l+1$ levels, and the number of entries in level i is N_i. With N_0 and the growth factor r as input parameters, the size of each level can be calculated as $N_i = N_0 \cdot r^i$. The total number of entries in the store is $N = \sum N_i$. The type of index used by level i is denoted as t_i. For the index of type t_i, its memory cost in bits per entry is denoted as $C_M[t_i]$. $C_{PR}[t_i]$, $C_{NR}[t_i]$, and $C_W[t_i]$ denote the cost

of a positive read, the cost of a negative read and the cost of write in disk access per operation. We also assume that the ratio of different operations in the workload are known beforehand: the ratio of positive reads, negative reads, and writes in the workload are r_{PR}, r_{NR}, and r_W, respectively. By choosing different types of indexes for each level, the goal is to meet a memory constraint and reduce the overall I/O cost. The overall average cost for each type of operations can be summarized as below:

$$S_{PR} = \sum_{0 \le i < l} \frac{N_i}{N} \times (C_{PR}[t_i] + \sum_{0 \le j < i} C_{NR}[t_j])$$

$$S_{NR} = \sum_{0 \le i < l} C_{NR}[t_i]$$

$$S_W = \sum_{0 \le i < l} \frac{N_i}{N} \times \sum_{0 \le j \le i} C_W[t_j]$$

Therefore, the average I/O cost of a random operation within a particular workload is:

$$C = r_{PR} \times S_{PR} + r_{NR} \times S_{NR} + r_W \times S_W$$

With a memory budget of M bytes, the constraints for this optimization problem are:

$$\sum N_i * C_M[t_i] \le M, 0 \le t_i \le 7$$

By using a heuristic search, the optimal value can be easily found for the above optimization problem.

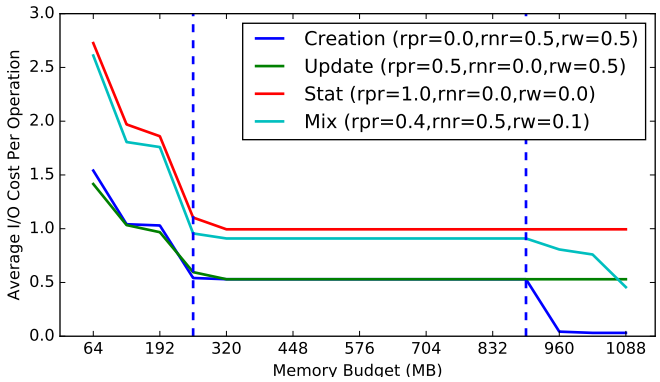


Figure 5.9: The per-operation cost estimated by the model.

Figure 5.9 shows the average cost of a key-value operation under different file system metadata workloads and memory constraints. In this figure, $l = 5$, $r = 8$, and $N_0 = 2^{17}$. So the key-value store has about a half billion entries in total. The figure shows four file system metadata workloads as an example: file creation in an empty file system (creation), updating inode attributes (update), querying inode attributes (stat), and a mix of reads and writes (mix). The ratio of key-value operations is calculated by designating file metadata operations into read and write operations. For example, since the creation workload creates files from an empty file system, all existence checks are negative reads, which means that $r_{NR} = 0.5$ and $r_W = 0.5$. From Figure 5.9, we can see that the average cost gradually decreases as the memory budget increases. For a creation workload, when the memory budget allows the key-value store to cache a filter at each level, the creation cost reaches the lowest point. For other workloads dominated by positive reads, one disk read is the lower bound. When the memory budget is between 256MB and 900MB, the four workloads use the same layout: the first four levels use a stepped-merge layout, three-level SSTable indexes and multi-level cuckoo filters; level 4 has only one sub-level with a three-level SSTable index; and level 5 has one sub-level without caching any additional index and filter. The multi-store layout is illustrated in Figure 5.4. Traversing down the level hierarchy, the memory cost of the index decreases from the multi-level cuckoo filter to not caching any index at all.

5.5 Evaulation

Using macro- and micro-benchmarks, we evaluate SlimFS's overall performance and explore how its system design and algorithms contribute to meeting its goals. We specifically examine (1) the performance of SlimFS's in-memory indexing data structures in isolation; and (2) an end-to-end evaluation of SlimFS's throughput, memory overhead, and latency.

5.5.1 Evaluation System

All our experiments are evaluated on a Linux desktop configured as is listed in Table 5.3.

We compare performance of SlimFS layered on top of SlimDB, against TableFS layered upon LevelDB, RocksDB [Hyp13a], and HyperLevelDB [Hyp13b]. Both RocksDB and HyperLevelDB are forks of the original LevelDB with improvements

Linux	Ubuntu 12.10, Kernel 3.5.0 64-bit version
CPU	Intel Core 2 Quad Processor Q9550 2.83 GHz
DRAM	4GB DDR SDRAM
Solid State Disk	Intel 520 Solid State Drive 120GB SATA II 2.5in with 110GB effective space Random Read 4,000 IO/sec peak Random Write 2,500 IO/sec peak Sequential Reads 245 MB/sec Sequential Write 107 MB/sec

Table 5.3: Hardware configuration for experiments.

on compaction and solid-state disks. All key-value stores are configured to use filters at 16 bytes per key. The growth factor for all key-value stores is 8, the size limit for Level-0 is 20 SSTables, and the size of each SSTable is 32MB.

All tested systems use Ext4 as the underlying file system to store their SSTables. Ext4 is mounted with "ordered" journaling to force all data to be flushed out to disk before its metadata is committed to disk. The write ahead logs of all tested systems are asynchronously committed to disks every 5 seconds. As in the evaluation for TableFS, SlimFS also pads its inode attributes to 256 bytes. The macro-benchmarks are evaluated on top of solid-state devices. Unlike magnetic hard disks, a solid-state disk provides a lot more random I/Os. Its basic random I/O unit (a page) can be considered to be 4KB, which means that B in the previous asymptotic analysis equals 16, assuming that the size of each entry (inode) is 256 bytes. The performance of small writes and reads in SSDs are asymmetric, since updating a single page requires first erasing an entire erase block of pages and then writing the modified block in its entirety. Since flash memory can undergo only a limited number of erase cycles before it fails, write amplification is therefore an important metric to examine.

5.5.2 Full System Benchmark

In this section, we report experiments done with an upper level file system driving an underlying KV-store. Our benchmark consists of two phases: a create phase initial-

izing the file system namespace, and a query phase reading/writing file attributes. The file system namespace in each experiment features a set of directories with each containing 128 empty files.

The reason we choose a simple file system namespace is to avoid measuring the effectiveness of directory entry caching on the path lookup resolution. Our benchmark application first generates all the directories randomly, and then creates all files by picking a random empty directory each time. In this workload pattern, the parent directories of newly created files are cached in the directory entry cache before their creation. Thus file and directory creation operations in this workload actually issue one lookup as an existence check and one insertion into the underlying key value store. There are 440 million empty files in total, and $4.4 \times 10^9/128 \approx 34 \times 10^6$ directories. The total size of the inodes of all entries is about 104 GB, which almost saturates the solid-state disk used in the evaluation. About 6GB space is reserved for the metadata, write-ahead logs, and other required data structures used in each KV-stores.

The query phase issues metadata operations `stat` on randomly selected files. Picking up files randomly limits file system's internal metadata cache effect to insignificance, provided the size of cache is much smaller than the size of the namespace. By randomly picking files following a uniform distribution, the benchmark application can get rid of the caching effect of the directory entry cache when the cache size is much smaller than the number of directories. As such, `stat` operations on a random file are generally translated into two lookup operations in the underlying key value store.

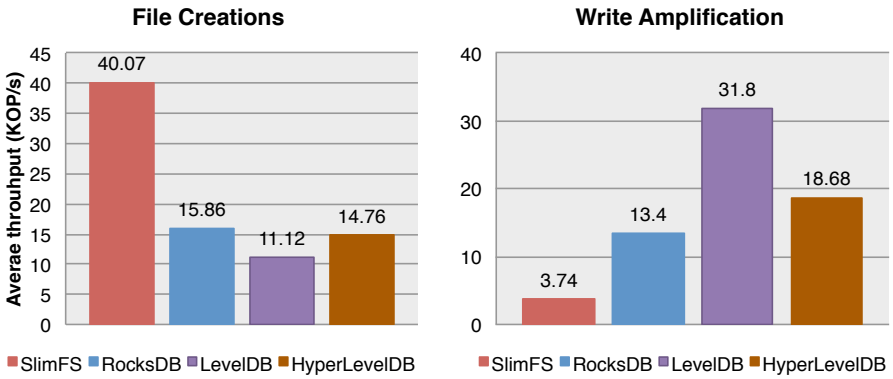


Figure 5.10: *The average file creation throughput and write amplification of tested system over the entire create phase.*

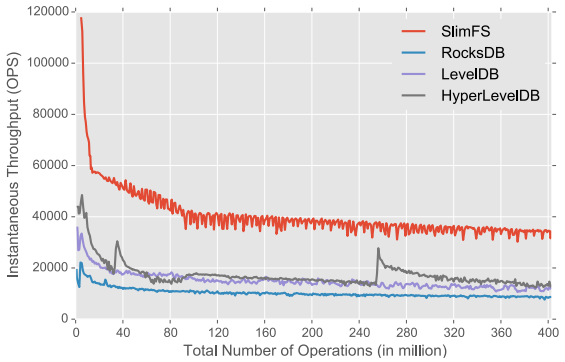


Figure 5.11: This figures shows the instantaneous file creation throughput during the create phase. Each data point shows the average throughput within 100 seconds time window, and is sampled when every 1 million new files are created.

Insertion Performance: Figure 5.10 (a) lists the average file creation throughput and write amplification over the entire create phase. Since the create phase starts with an empty file system and files are created without name collisions, most file existence checks avoid reading the actual disk with the help of LSM-tree's in-memory filters. The creation phase measures the random insertion performance of the underlying key value stores. SlimFS shows higher throughput than TableFS with different stores. The write amplification is calculated as the ratio between the actual amount of written data seen by the device and the total number of files. The write amplifications shown in Figure 5.10 (b) match well to their theoretical bounds. SlimFS has 4 levels and its write amplification is 3.74, which is very close to $\log_r N = 4$, the theoretical bound of the Stepped-Merge algorithm. Other key value stores have much higher write amplification. For example, the write amplification of LevelDB, which is closest to the standard LSM-tree is 31.8 matching its theoretical bound $r \log_r N = 32$.

Figure 5.11 shows the instantaneous throughput of each file system during the create phase. Each data point shows average file creation throughput during a 100-second time window. The data points are sampled once 1 million new files are created. In general, SlimFS is 2 to 3 times faster than other tested systems. The throughput of all systems gradually slow down as the insertion cost grows when more entries are inserted. The throughput variance of SlimFS is higher than other systems because SlimFS's compaction procedure tend to select more SSTables to compact in each compaction pass.

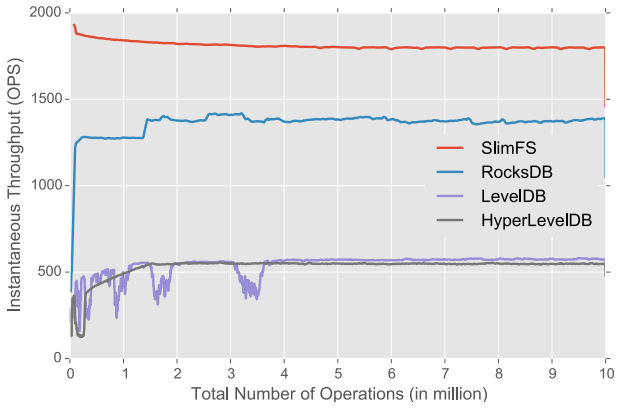


Figure 5.12: This figures shows the instantaneous file `stat` throughput during the query phase. Each data point shows the average throughput within a 10-second time window, and samples are taken when every 10,000 file `stat` requests are issued.

Stat Workloads: Figure 5.12 plots the `stat` throughput. The query phase performs 10M operations in total. As can be seen from the figure, there are two distinct stages in this phase for all systems except SlimFS. The `stat` throughput fluctuates in the first stage, and then reaches its steady phase in the second stage.

One factor that causes this behavior is that all tested file systems have to be rebooted to release all cached information before running the query phase. SlimFS aggressively loads all metadata indices (cuckoo filters and SSTable index) into memory, while other key value stores load their indexes on the fly. Since filenames of `stat` requests are uniformly distributed, it takes a long time for other key value stores to warm up their metadata index cache. Another more important factor is that compaction procedures in all other key value stores will be triggered if some SSTables have been frequently read. The compaction procedure will try to reduce the number of levels in the key value store if the workload becomes read intensive. However, even when all other tested systems reach the steady phase, the throughput of SlimFS is still two times faster than these alternatives. SlimFS's throughput is around 91% of one SSD's raw read throughput in terms of the number of 4 KB-blocks read per second. This is because the compact SSTable three-level index saves one disk read for every lookup operation into the key value store. With multi-level cuckoo filter, every lookup operation only needs to read one SSTable. So when all the metadata indexes can be cached in memory, the I/O cost of each lookup is exactly 1.

Figure 5.13 further demonstrates the evidence of improved tail latency by using compact index in SlimFS. Figure 5.13 shows the distribution of latencies of the `stat`

operation for each system. For SlimFS, its 99.9 percentile latency is 0.6ms, which is significantly better than other tested systems. The tail latencies in other tested systems are affected by the concurrent compaction procedure.

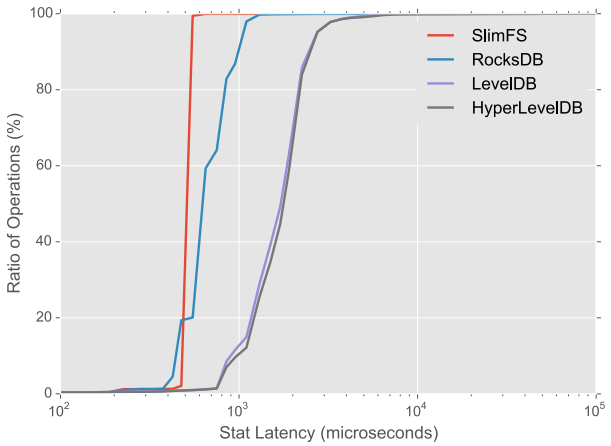


Figure 5.13: The latency distribution of `stat` operations for all tested systems during the query phase.

5.5.3 Compact SSTable Index Microbechmark

This section demonstrates the effectiveness of compacting the index for each SSTable. The experiments compare LevelDB's original SSTable index against our three-level indexing strategy. Analysis of the experiments results focuses on two main metrics: the memory cost and lookup latency per key. It is expected that the enhanced three-level index uses less memory but incurs additional costs on lookup for decompression.

Experiment Design: In the first experiment, an empty SSTable is first filled according to given key distribution for each case; the second stage of this experiment opens this newly-created SSTable, loads its index block into the memory to obtain its memory consumption, then generates 1 million random queries to measure average lookup latency. In the original LevelDB, each SSTable consists of a list of data blocks and a single index block. Other types of blocks holding information such as a bloom filter or table metadata are excluded.

In all experiments, an SSTable has a fixed size of 32MB and each data block has a size of 4KB. All inserted key-value entries have a fixed size of 256 bytes. With table formatting and key prefix compression provided by original LevelDB, SSTables

generated in the experiments (with or without compact index) have 148639 entries and 8744 data blocks, with each block on average holding approximately 17 entries. Four different prefix group size distribution patterns are used to evaluate the index: each prefix group has either fixed 16, 32, or 64 entries sharing the same prefix, or has a Zipfian distributed prefix group size with a maximum of 8000 entries. According to the experimental results, these different patterns generated 9290, 4645, 2323, and 26 distinct prefix groups within a single SSTable, respectively.

Memory Consumption: Table 5.4 shows the experimental results in terms of memory consumption between SlimDB and LevelDB. LevelDB's default indexing mechanism is built upon a sorted array of the last keys of each data blocks. Experiments show that this mechanism can take up to almost 18 bits per key in order to store the entire index in the memory. However, if key compression is applied to LevelDB — which means storing only the unique key prefix that can distinguish the last key of a data block from the first key of the next data block, instead of storing the entire key — LevelDB's memory consumption can be reduced from 18 to about 10.5 bits per key. LevelDB also assumes that each data block has a variable and unpredictable size. So in the indexing structure, LevelDB stores the offset and length of each data block in order to later locate and read those data blocks. In SlimDB design, all data blocks have a fixed size, which allows it to skip storing block locations within the index. With this assumption, LevelDB's memory consumption can be reduced from 10.5 to about 8 bits per key, which we see as the best memory usage that can be achieved from an LevelDB style array-based indexing mechanism. In contrast, SlimDB's compact three-level indexing only requires 1.5 to 2.5 bits per key to represent the entire index, which is as little as 8% to 14% of the memory space needed by LevelDB out of the box. Another observation from Table 5.4 is that the memory savings depend on workload patterns. SSTables with a lot of small prefix groups require relatively larger memory footprints compared to SSTables storing only a few large prefix groups. This is because a larger set of distinct prefix groups leads to a larger prefix ECT structure is used as the first level of the three-level index.

Lookup Performance: Table 5.5 shows the experimental results in terms of in-memory lookup throughput against these indexes. As can be seen from the table, the three-level indexing mechanism has a longer lookup time, — about 5 to 7 times slower than the original LevelDB index. This is because our more compact indexing requires a more sophisticated search algorithm to complete each query. A closer analysis found that decoding the first-level index consumes 70% of the CPU cycles for fixed prefix group sized workloads. For Zipfian-prefix-group size workloads, decoding the third-level index occupies 60% of the CPU cycles. Compact three-level

	Fix.16	Fix.32	Fix.64	Zipf
SlimDB	2.56	1.94	1.57	1.56
LDB_FixedBlock	7.58	7.82	8.18	8.43
LDB_KeyCompress.	10.37	10.61	10.98	11.23
LDB_Default	17.39	17.39	17.39	17.39

Table 5.4: The memory consumption of different SSTable indexes measured as bits per key, for various patterns of prefix groups (3 fixed sized and a Zipfian distribution

indexing trades greater CPU cost for saving memory space, which is worthwhile when the gap between CPU resources and memory resources is large. On the other hand, reducing memory costs allows the key-value store to cache more indexes, which can avoid unecessary disk reads.

	Fix.16	Fix.32	Fix.64	Zipf
SlimDB (KOP/s)	147.5	143.6	149.5	288.5
LevelDB (KOP/s)	1042.7	1019.4	1016.3	1046.0

Table 5.5: Average SSTable index lookup speed for SlimDB and the original LevelDB in thousands of lookups per second.

5.5.4 Multi-Level Cuckoo Filters Microbenchmark

While the high throughput of flash disks often limits the CPU budget available for in-memory indexing, this section demonstrates the computation efficiency of our multi-level cuckoo filters. The multi-level cuckoo filter is compared against traditional ways of using cuckoo filters in LevelDB and other key-value stores. In this section, the multi-level cuckoo filter is denoted as "MLCF", and the cuckoo filter is denoted as "CF".

Experiment Design: We focused on the bulk insertion and lookup speed of multi-level cuckoo filters. All experiments are single-threaded programs.

The benchmark builds a filter for an 8-sub-level key-value store. Each sub-level has 10M random 16-byte keys. This micro-benchmark involves only in-memory accesses (no flash I/O). Keys are pre-generated, sorted, and passed to our filters. There is also a parameter d called "duplication ratio" that controls the ratio of duplicated

keys at each level. If $d = 10\%$, this means that for any level i ($0 \leq i < 7$), 10% of the keys are selected from the first 10% of keys in level 8, and the other 80% of keys are distinct from all other keys in the key-value store. Later, we show the impact of this ratio of duplicated keys on the performance of in-memory filters.

To measure lookup performance, we use both true-positive queries and false-positive queries. For true-positive queries, the benchmark issues 80M random requests on positive keys, meaning that the keys are present in the key-value store. For false-positive queries, the benchmark issues 80M random requests on negative (not present) keys.

Space Efficiency and Achieved False Positive Rate: In this experiment, CF is configured to use a 16-bit hash fingerprint. MLCF uses a 14-bit hash fingerprint and a 3-bit value field to index the level number. Table 5.6 shows the actual memory cost per key and the achieved false positive rate for these configurations. The actual memory cost includes the space inflation caused by a less than 100% load factor for the cuckoo hashing table, as well as such as the secondary table used by MLCF. By comparing the memory cost of MLCF and CF, the memory overhead introduced by the secondary table is negligible because the false positive rate is low.

	CF	MLCF
RAM Cost (Bits/Key)	16.78	16.67
False Positive Rate	0.001	0.002

Table 5.6: RAM usage and false positive rate of different filters.

Insert Performance: Table 5.7 shows the bulk insertion performance of different filter implementations under different duplication ratios. CF is faster than MLCF in all cases. This is because MLCF has to check whether an entry has already been inserted into other levels when inserting a new entry. When the duplication ratio becomes large, the average insertion throughput of MLCF becomes higher. The insertion speed of Cuckoo hash tables is higher when its table occupancy is lower. Since MLCF only uses a single table to store all entries, a high duplication ratio leads to low table occupancy.

Lookup Performance: Figure 5.14 shows the lookup performance of different filter implementations under different duplication ratios. MLCF is faster than CF in all cases. Since there is only a single hash table in MLCF, the lookup operation requires fewer memory references than the other two alternatives. When the duplication ratio grows larger, the table occupancy in all three filters becomes lower and

Duplication Ratio	0%	10%	30%	50%	70%	90%
Cuckoo Filter	2.0	2.05	2.05	2.05	2.05	2.05
Multi-level CF	0.86	0.90	0.98	1.11	1.31	1.40

Table 5.7: Bulk insertion throughput of three filters under different duplication ratio in millions of inerstions per second.

therefore, all three filters gain higher lookup throughput. Under a higher duplication ratio, most entries in CF are stored in the higher level, so the lookup procedure can find these entries earlier. However, the duplication ratio does not affect the performance of negative queries in CF. This is because the lookup procedure still needs to check each level. A single MLCF has better average lookup performance compared to CF in multiple levels.

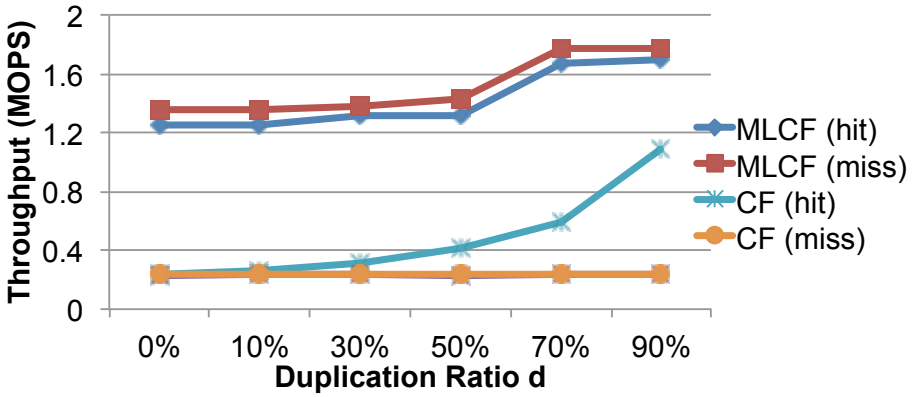


Figure 5.14: Average lookup throughput of three filters under different duplication ratios.

5.6 Summary

General-purpose LSM-tree implementations usually lack optimizations on read and write amplification for specific key-value workloads running on solid-state disks. We present techniques that allow key-value workloads with semi-sorted data and non-blind writes to run more efficiently, in terms of both I/O activities and memory

consumption. To improve read performance, two ideas for shrinking an index are proposed: one is a three-level compact index that only costs 1.9 bits per key to locate the block position of a particular key inside the SSTable; the other is the design of a multi-level cuckoo filter that not only bounds the worst-case disk reads of lookup operations, but also improves their average latency. Through the integration with SlimFS, experiments show that our compact indexes, combined with write-optimized stepped-merge algorithm, can achieve a better balance between read and write amplification, and outperforms many existing LSM-tree implementations by several times. While aiming at file systems, some of our techniques such as multi-level cuckoo filter and the analytical model can also be applied to other workloads that require only semi-sorted key order and non-blind writes.

Chapter 6

IndexFS: Metadata Management For Distributed File Systems

As discussed in Chapter 2, lack of a highly scalable and parallel metadata service is becoming an important performance bottleneck for many distributed file systems in both the data intensive scalable computing (DISC) world [HDF] and the high performance computing (HPC) world [Day08, New08]. This is because most cluster file systems are optimized mainly for scaling the data path (i.e., providing high bandwidth parallel I/O to files that are gigabytes in size) and have limited metadata management scalability. They either use a single centralized metadata server, or a federation of metadata servers that statically partition the namespace (e.g., Hadoop Federated HDFS [HDF], PVFS [RL06], Panasas PanFS [WUA+08], and Lustre [Lus]). Limited metadata scalability also handicaps massively parallel applications that require concurrent and high-performance metadata operations.

To fix these problems, one goal of my thesis work is to design a scalable metadata middleware that can scale metadata performance of existing distributed file systems used in data-center environment. We have built a solution called IndexFS that scales metadata management for existing file systems from both horizontal scaling and vertical scaling aspects. Dynamic namespace partitioning and storm-free caching techniques are used to scale out metadata management across many machines. Integration with out-of-core metadata representation, introduced in Chapter 4, and bulk insertion are proposed into improve the performance of single metadata server. Since the architecture of the targeted distributed file system separates the processing of metadata and data for high throughput, the layering design is able to reuse the parallel data path to enhance the throughput of metadata operations.

This chapter will discuss these techniques in details and also compare the proposed system with other alternative file system metadata service designs.

6.1 IndexFS System Design

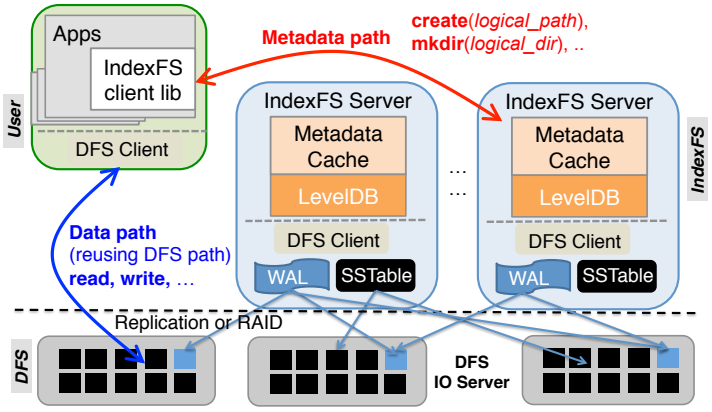


Figure 6.1: The IndexFS metadata system is middleware layered on top of an existing cluster file system deployment (such as PVFS or Lustre) to improve metadata and small file operation efficiency. It reuses the data path of the underlying file system and packs directory entries, file attributes and small file data into large immutable files (SSTables) that are stored in the underlying file system.

IndexFS is middleware inserted into existing deployments of cluster file systems to improve metadata efficiency while maintaining high I/O bandwidth for data transfers. Figure 6.1 presents the overall architecture of IndexFS. The system uses a client-server architecture:

IndexFS Client: Applications interact with the IndexFS middleware through a library directly linked into the application, through the FUSE user-level file system [fus], or through a module in a common library, such as MPI-IO [Cea96]. Client-side code redirects applications' file operations to the appropriate destination according to the type of operation. Metadata requests (e.g., `create` and `mkdir`), and data requests on small files with size less than 64KB (e.g., `read` and `write`), are handled by the metadata indexing module that sends these requests to the appropriate IndexFS server. For data operations on large files, client code redirects read requests directly to the underlying cluster file system to take full advantage of parallel I/O bandwidth.

A newly created but growing file may be transparently reopened in the underlying file system by the client module. When a large file is reopened in the underlying file system for write, some of its attributes (e.g., file size and last access time) may change relative to IndexFS's per-open copy of the attributes. The IndexFS server will capture these changes on file close using the metadata path. IndexFS clients employ several caches to enhance performance for frequently accessed metadata such as directory entries, directory server mappings, and complete subtrees for (writeback) bulk-insertion. Details about these caches will be discussed in later sections.

IndexFS Server: IndexFS employs a layered architecture as shown in Figure 6.1. Each server owns and manages a non-overlapping portion of file system metadata, and packs metadata and small file data into large flat files stored in the underlying shared cluster file system. File system metadata is distributed across servers at the granularity of a subset of a directory's entries. Large directories are incrementally partitioned when their size exceeds a threshold. Similar to TableFS, IndexFS packs metadata and small file data into large immutable sorted files (SSTables) by using the log-structured merge (LSM) tree [OCGO96]. Since LSM trees convert random updates into sequential writes, they greatly improve performance for metadata creation intensive workloads. Optimization techinques proposed in SlimFS can be also applied to the local storage of IndexFS. However, since SlimFS is proposed after IndexFS, experiments in this section do not include any optimization from SlimFS. For durability, IndexFS relies on the underlying distributed file system to replicate or RAID encode the LSM tree's SSTable files and write-ahead logs. Details about fault tolerance techniques used in IndexFS are presented in Section 6.1.6.

6.1.1 Dynamic Namespace Partitioning

IndexFS uses a dynamic namespace partitioning policy to distribute both directories and directory entries across all metadata servers. Unlike prior works that partition the file system namespace based on a collection of directories that form a sub-tree [DH06, WBML06], IndexFS's namespace partitioning works at the directory subset granularity. Figure 6.2 shows an example of distributing a file system tree to four IndexFS metadata servers. Each directory is assigned to an initial metadata server when it is created. The directory entries of all files in that directory are initially stored in the same server. This works well for small directories (e.g., 90% of directories have fewer than 128 entries in many cluster file system instances [WN13]) since storing directory entries together preserves locality for scan operations such as `readdir`. The initial server assignment of a directory is done through random

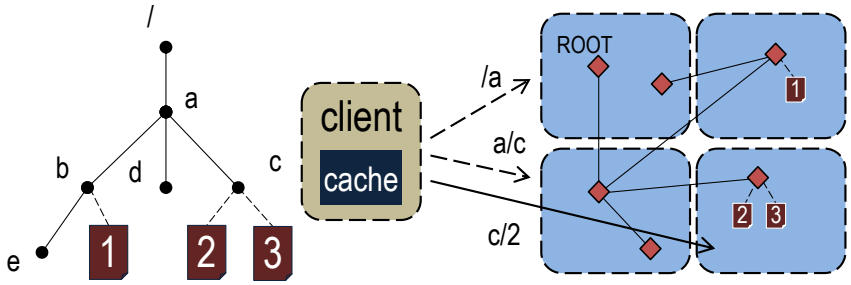


Figure 6.2: This figure shows how IndexFS distributes a file system directory tree evenly into four metadata servers. Path traversal makes some directories (e.g. the root directory) more frequently accessed than others. Thus stateless directory caching is used to mitigate these hot spots.

server selection. To reduce the variance in the number of directory entries stored in metadata servers, IndexFS also adapts the "power of two choices" load balancing technique [Mit01] to the initial server assignment. This technique assigns each directory by probing two random servers and placing the directory on the server with fewer stored directory entries. To reduce the number of probes, the metadata server can cache the number of directories store on each server, and update these numbers less frequently.

For the few directories that grow to a large number of entries, IndexFS uses the GIGA+ binary splitting technique to distribute directory entries over multiple servers [PG11]. Each directory entry is hashed to uniformly map it onto a large hash-space that is range partitioned. GIGA+ incrementally splits a directory in proportion to its size: a directory starts small, on a single server that manages its entire hash-range. As the directory grows, GIGA+ splits the hash-range into halves and assigns the second half of the hash-range to another metadata server. As these hash-ranges gain more directory entries, they can be further split until the directory is using all metadata servers. This splitting stops after each server owns at least one partition of the distributed directory. IndexFS servers maintain, and clients opportunistically cache, a partition-to-server mapping to locate entries of distributed directories. These mappings are inconsistently cached at the clients to avoid cache consistency traffic; stale mappings are corrected by any server inappropriately accessed [PG07, PG11].

6.1.2 Stateless Directory Caching

To implement POSIX file I/O semantics many metadata operations are required for each ancestor directory to perform pathname traversal and permission checking. This requires many RPC round trips if each check must find the appropriate IndexFS server for the directory entry subset that should contain this pathname component. The GIGA+ algorithm used by IndexFS removes almost all RPC round trips associated with finding the correct server by caching mappings of directory partitions to servers that tolerate inconsistency; stale mappings may send some RPCs to the wrong server, but that server can correct some or all of the client's stale map entries [PG11]. By using an inconsistent client cache, servers never need to determine which clients contain correct or stale mappings, eliminating the storms of cache updates or invalidation messages that occur in large scale systems with consistent caches and frequent write sharing [LCL+09, Sch03, VSK+03].

Once the IndexFS servers are known, there is still a need for RPCs to test existence and permissions for each pathname component because the original GIGA+ algorithm caches only server locations. This access pattern is not well balanced across metadata servers because pathname components near the top of the file namespace tree are accessed much more frequently than those lower in the tree (see Figure 6.2) due to the nature of pathname resolution. The pathname has to be resolved following the top-down order on pathname components, that is, ancestor directories have to be resolved before their children components. To reduce lookups of pathname components, IndexFS maintains a consistent client cache of pathname components and their permissions (but not their attributes) without incurring invalidation storms by assigning short term leases to each pathname component offered to a client and delaying any modification until the largest lease expires. This allows IndexFS servers to record only the largest lease expiration time with any pathname component in its memory and not per-client cache states. The server pins the entry in its memory and blocks updates until all leases have expired. If the cache is filled up, then the server can either refuse to offer a new lease, or delay offering the lease until some lease expires that releases the cache resources. This requires a small amount of additional IndexFS server state (only one or two variables for each directory entry) and it does not cause invalidation storms.

Any operation that wants to modify the server's copy of a pathname component, which is a directory entry in the IndexFS server, blocks operations that want to extend a lease (or returns a non-cacheable copy of the pathname component information) and waits for outstanding leases to expire. Although these mutation opera-

tions may incur higher latency, client latency for non-mutation operations, memory and network resource consumptions are greatly reduced. This method assumes the clock on all machines are synchronized, which is commonly achievable in modern data centers [BMK10, CDE+12]. A typical optimization to reduce blocking duration and speed up invalidations that are not part of a storm would be to record up to N (a small number) client IDs with each lease and send explicit invalidations provided the number of leases does not exceed N.

Several policies have been investigated for the lease duration for individual cached entries. The simplest is to use a fixed time interval (e.g., 200ms) for each lease. However, some directories, such as those at the top of the namespace tree, are frequently accessed and unlikely to be modified, so the lease duration for these directory entries benefits from being extended. IndexFS's non-fixed policies use two indicators to adjust the lease duration: one is the depth of the directory tree (e.g., $3sec/depth$), and the other is the recent read to write (mutation) ratio for the directory entry. This ratio is measured only for directory entries cached in the metadata server's memory. Because newly created/cached directory entries do not have an access history, the lease duration $L/depth$ is set where $L = 3s$ in the evaluation experiments. For directory entries that have history in the server's memory, a exponential weighted moving average (EWMA) is used to estimate the read and write ratio [ewm]. Suppose that r and w are the recent counts of read and write requests respectively, then the offered lease duration is $\frac{r}{w+r} \cdot L_r$, where $L_r = 1s$ in the evaluation experiments. This policy ensures that read-intensive directory entries will get longer lease durations than the write-intensive directory entries.

Server-side Metadata Caching: Instead of keeping a client cache of pathname components, an alternative approach is to maintain the cache on the metadata servers. Each metadata server can serve as a proxy for pathname resolution. So the clients can consult any metadata server to parse a pathname and find out the actual server location of the object pointed by the pathname. This design is similar to ShardFS [Xia13], which will be discussed later in Section 6.2. The difference is that ShardFS replicates the directory lookup information in each metadata server, while this approach only keeps a temporary copy of the information that can be invalidated at any time. For a cluster with only a few metadata servers, the time-based lease is not necessary and the traditional lease will not incur an invalidation storm.

Compared to the client caching approach, the server side metadata caching has several advantages: 1) It is more reliable to only use leases on the server side and keep no states in clients. Compared to the server programs, the client libraries are easier to become corrupted or malfunction due to other user codes. This approach also

removes dependency on time synchronization across machines. 2) It may also balance resource utilization on metadata servers by caching popular items in randomly selected servers.

The disadvantages are as follows: 1) Server-side metadata caching does not utilize all computing and memory resources available in the client machines. 2) The metadata server picked for path resolution and the server storing the actual object (indicated by the path) may not be the same server. There is a high probability that each metadata operation requires at least two RPCs. If the client's access pattern shows strong locality, client caching may only need one RPC for each metadata operation.

6.1.3 Integration with Log-Structured Metadata Storage

The IndexFS metadata storage backend integrates with the modified TableFS to manage metadata and small files locally. In order to integrate with IndexFS, we have modified the data schema inside TableFS to support directory splitting, and also extended TableFS's functionality to support bulk insertion. The log-structured storage format improves the single node metadata performance, especially for write intensive workloads. The following sections will discuss how IndexFS integrates with TableFS and the details about the TableFS modifications.

Metadata Schema: Similar to the prior work on TableFS, IndexFS embeds inode attributes and small files with directory entries and stores them into a single LSM tree with an entry for each file and directory. We have modified the data schema used by TableFS to support splitting directories and the GIGA+ algorithm. As shown in Table 6.1, a new field called partition ID is added in the key of each entry. This partition ID is used to specify which partition the entry belongs to so that IndexFS can verify the entries when splitting or bulk inserting directory partitions. The value of a directory entry contains directory partition mapping information along with standard metadata attributes. The mapping information is necessary for identifying the server location of every directory partition. For large files, the file data field in a file row of the table is replaced by a symbolic link pointing to the actual file object in the underlying distributed file system.

Partition Splitting and Migration: IndexFS uses a faster technique for splitting a directory partition than is used by GIGA+. The immutability of SSTables in LevelDB makes fast bulk insertion possible – an SSTable whose range does not overlap any part of a current LSM tree can be added to LevelDB (as another file

key	*Parent directory ID, Partition ID, Hash(Name)*
value	*Name, Attributes, Mapping|File Data|File Link*

Table 6.1: The schema of keys and values used by IndexFS. Only the value of a directory contains the "mapping" data, which is used to locate the directory partition's server.

at level 0) without its data being pushed through the write-ahead log, in-memory cache, or compaction process. To take advantage of this opportunity, we extended LevelDB to support a three-phase directory partition split operation:

- Phase 1: The server initiating the split locks the directory (range) and then performs a range scan on its LevelDB instance to find all entries in the hash-range that needs to be moved to another server. Instead of packing these into an RPC message, the results of this scan are written in SSTable format to a file in the underlying distributed file system.

- Phase 2: The split initiator sends the path to the SSTable-format split file the split receiver in a small RPC message. Since this file is stored in shared storage, the split receiver directly inserts it as a symbolic link into its LevelDB tree structure without actually copying the file. The insertion of the file into the split receiver is the commit part of the split transaction.

- Phase 3: The final step is a clean-up phase: after the split receiver completes the bulk insert operation, it notifies the initiator, who deletes the migrated key-range from its LevelDB instance, unlocks the range, and begins responding to clients with a redirection to files in this range.

For the column-style storage schema, only index tables need to be extracted and bulk inserted at the split receiver. Data files, stored in the underlying shared distributed file systems, can be accessed by any metadata server. In the current implementation, there is a dedicated background thread that maintains a queue of splitting tasks to throttle directory splitting so only one split occurs at a time. This is a simple way to reduce lock conflicts caused by multiple concurrent splits and migitate the variance in throughput and latency experienced by clients.

6.1.4 Metadata Bulk Insertion

Even with scalable metadata partitioning and efficient on-disk metadata representation, the IndexFS metadata server can only achieve about 10,000 file creates per second in the testbed cluster. This rate is dwarfed by the speed of non-server based systems such as the small file mode of the Parallel Log Structured Filesystem (PLFS [BGG+09]) which can achieve millions of file creates per second [TB13, FBZ+14]. Inspired by the metadata client caching and bulk insertion techniques we used for directory splitting, IndexFS implements write back caching at the client for creation of new directory subtrees. This technique may be viewed as an extension of Lustre's directory callbacks [Sch03]. By using bulk insertion, IndexFS strives to match PLFS's create performance.

Since metadata in IndexFS is physically stored as SSTables, IndexFS clients can complete creation locally if the file is known to be new, and later bulk insert all the file creation operations into IndexFS using a single SSTable insertion. This eliminates the one-RPC-per-file-create overhead in IndexFS, allowing new files to be created much faster and enabling total throughput to scale linearly with the number of clients instead of the number of servers. To enable this technique, each IndexFS client is equipped with an embedded metadata storage backend library that can perform local metadata operations and spill SSTables to the underlying shared file system. As IndexFS servers are already capable of merging external SSTables, support at the server-side is straightforward.

Although client-side writeback caching of metadata can deliver ultra high throughput bulk insertion, global file system semantics may no longer be guaranteed without server-side coordination. For example, if the client-side creation code fails to ensure permissions, the IndexFS server can detect this as it first parses an SSTable bulk-inserted by a client. Although file system rules are ultimately enforced, error status for rejected creates will not be delivered back to the corresponding application code at the `open` call site, and could go undetected in error logs. Quota control for the (tiny fraction of) space used by metadata will be similarly impacted, while data writes to the underlying file system can still be growth limited normally.

IndexFS extends its lease-based cache consistent protocol to provide the expected global semantics. An IndexFS client wanting to use writeback caching and bulk insertion to speed up the creation of new subtrees issues a `mkdir` with a special flag "`LOCALIZE`", which causes an IndexFS server to create the directory and return it with a renewable write lease. During the write lease period, all files (or subdirectories) created inside such directories will be exclusively served and recorded by the

client itself. Before the lease expires, the IndexFS client must return the corresponding subtree to the server, in the form of an SSTable, through the underlying cluster file system. After the lease expires, all bulk inserted directory entries will become visible to all other clients. While the best creation performance will be achieved if the IndexFS client renews its lease many times, it may not delay bulk insertion arbitrarily. If another client asks for access to the localized subtree, the IndexFS server will deny future write lease renewals so that the writing client needs to complete its remaining bulk inserts quickly. If multiple clients want to cooperatively localize file creates inside the same directory, IndexFS `mkdir` can use a "`SHARED_LOCALIZE`" flag, and conflicting bulk inserts will be resolved at the servers arbitrarily (but predictively) later. As bulk insertion cannot help data intensive workloads, IndexFS clients automatically "expire" leases once significant data writing is detected.

Inside a localized directory, an application is able to perform all metadata operations. For example, `rename` is supported locally but can only move files within the localized directory. Any operation not compatible with localized directories can be executed if the directory is bulk inserted to the server and its lease expired.

6.1.5 Rename Operation

The namespace distribution in IndexFS increases the complexity of implementing `rename` operations. `rename` is usually decomposed into two primitive metadata operations: the removal of the old object and creation of the new object identical to the old one. Since the file system namespace in IndexFS is distributed across many servers, the two objects may be located in two different places, and therefore a distributed transaction is often required to guarantee correct behavior. Directory renaming is more complicated than file renaming because concurrent directory `rename` operations may interfere with each other even they do not work on the same objects. It often requires multiple locks on the ancestor directories to prevent an orphaned loop [DH06]. The following text discusses the details of implementing `rename` in IndexFS and the locking strategy for avoiding conflicts between concurrent `rename` operations.

Atomicity of Rename Operation: Because `rename` consists of two stages (removal and creation), IndexFS has to ensure its atomicity such that the two stages either succeed or fail altogether. It should not corrupt the file system (i.e. by applying some partial changes), even when failure events happen during the operation.

Currently IndexFS adopts the two phase commit protocol (2PC) for `rename`, which involves coordinating multiple participants (servers). There are two different roles for coordination in 2PC: coordinator and participant. Both roles are designated to metadata servers instead of clients, since the clients are unreliable and difficult to track especially in a shared data center environment. The source server where the source directory resides is always chosen to be the coordinator for the transaction. Participants include the source server, the destination server and all the servers that manage any other ancestral directories that need to be locked during `rename` operation. When a user client issues a `rename` operation, it locates the coordinator server, and initiates the transactions. The coordinator will perform the standard 2PC procedure: the coordinator sends a message to each participant to ask them prepare necessary locks and push the changes into their logs; once the coordinator receives the agreement message from all participants, the coordinator sends a commit message to all participants to complete the operation and release all locks held during the transaction; when every participant acknowledges the completion of its work, the coordinator finalizes the `rename` transaction.

To provide recovery ability in a two phase commit transaction after failures, write-ahead loggings are used by both coordinator and participants to record the partial state of the transaction. As IndexFS has already used LevelDB for metadata storage and LevelDB itself utilizes write-ahead logs for durability, IndexFS directly pushes the states into LevelDB for rename operation.

Locking in Rename Operation: A correct implementation of `rename` operation also requires locking multiple locks to avoid interference from concurrent operations, otherwise it may cause corrupt states in the file system. Figure 6.3 shows how two concurrent `rename` operations result in an orphaned loop in the file system namespace. In figure 6.3, there are 7 directories and two concurrent `rename` operations: one is to rename $/b/d$ to $/a/c/e/d$, and the other is rename $/a/c$ to $/b/d/f/c$. Without other proper locks, both rename operations will succeed in an interleave way. After both operations succeed, d becomes child directory of e, and c is the child of f, but all of them are disconnected from the top of the tree.

A naive solution is to take a global lock for all `rename` operations. Although we have shown that `rename` is an infrequent operations in Chapter 3, the single lock server may still become an performance bottleneck. The fine-grained locking approach is to take an exclusive writer lock on the source, the destination directories and their parent directories, as well as shared reader locks on the ancestor directories. Since the path resolution has already acquired read leases on every ancestor directory, these shared reader locks will not incur too much overhead. The reader

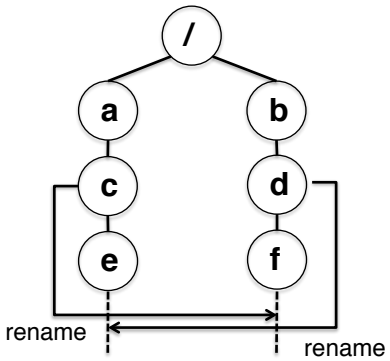


Figure 6.3: Orphaned loop from two `rename` operations.

locks will also allow other types of non-conflict metadata operations to perform concurrently without any stall. This locking strategy also prevents the formation of orphaned loops, since the reader lock ensures that the source and destination objects are always accessible to the root directory during the `rename` operation. When acquiring locks during the transaction, IndexFS servers always follow a global locking order (a lexicographical order on the pathname) to avoid any deadlock.

6.1.6 Fault Tolerance

IndexFS is designed as middleware layered on top of an underlying failure-tolerant and distributed file system. IndexFS's fault tolerance strategy is to push states into the underlying file system – large data into files, metadata into SSTables and recent changes into write-ahead logs (WAL). The IndexFS server processes are monitored by standby server processes that are prepared to replace failed server processes. Zookeeper, a quorum consensus replicated database, is used to store (as a lease) the location of each primary server [HKJR10]. Each IndexFS metadata server maintains a separate write-ahead log that records mutation operations such as file creates and renames. When a server crashes, its write-ahead log can be replayed by a standby server to recover consistent state.

Leases for client directory entry caching are not durable. A standby server restarting from logs blocks mutations for the largest possible timeout interval. The first lease for a localized directory should be logged in the write-ahead log so a standby server will be prepared for a client writing back its local changes as a bulk insert.

Some metadata operations, including directory splitting and rename operations, require a distributed transaction protocol. These are implemented as a two-phase distributed transaction with failure protection from write-ahead logging in source and destination servers and eventual garbage collection of resources orphaned by failures.

IndexFS supports two modes of write-ahead logging: synchronous mode and asynchronous mode. The synchronous mode group commits a number of metadata operations to disk to make them persistent. The asynchronous mode instead buffers log records in memory and flushes these records when a time (default 5 seconds) or size threshold (default 16KB) is exceeded. The asynchronous mode may lose data when a crash happens, but provides much higher ingestion throughput than the synchronous mode. Because most local file systems default to asynchronous mode, it is also the default setting in the experiments of the evaluation section.

6.2 Comparison of System Designs

There are also several recent works related to scaling file system metadata management. In this section, IndexFS is compared against two alternative designs: ShardFS and Giraffa, which also dynamically distributes metadata with fine granularity. That's because static subtree partitioning of the namespace cannot provide good load balancing when the workload only accesses a small portion of the namespace. The comparison of file system metadata designs focuses on key factors that affect system performance such as load balancing, throughput and latency of metadata operations.

As indicated in the previous section, pathname resolution is an important factor restricting the scaling of distributed metadata management. IndexFS scales pathname resolution by coherently caching namespace information (structure, names, permissions) in each client under the protection of a (leased) lock, and simplifies server error handling logic by blocking all mutations until all leases have expired. Caching state under coherent leases is a replication strategy with replication costs proportional to the number of clients and to the size of the working set for the cache (number of directories frequently consulted in a pathname lookup). ShardFS is an example system that uses full replication of the pathname resolution information in each metadata server. This avoids multiple RPCs for lookup, but increases the complexity of modification operations. An alternative approach, taken by Giraffa,

is to relax file system access control semantics and store the full pathname for each entry to reduce the number of lookups.

The following section will explain the design principles of the two alternative systems, and discuss the performance implications brought by these three design choices.

6.2.1 Table partitioned namespace (Giraffa)

As shown in Figure 6.4, Giraffa stores file system metadata inside a distributed table, HBase [Fou], which provides single-row transaction guarantees. Each file or directory in the Giraffa namespace is stored as one row inside a table in HBase. In order to maintain the hierarchical namespace of the file system, Giraffa embeds its file system tree structure inside the row keys of all the file system objects. The default strategy is to use as the key a full pathname prefixed with the depth (of this file system object) in the namespace tree. This ensures that all entries within a same directory can share the same prefix of their row keys, which secures the necessary locality required to implement `readdir` efficiently. Giraffa translates metadata operations into a set of key-value operations to HBase, and reuses the load balancing techniques and persistence guarantees provided by both HBase [Fou] and HDFS [HDF].

Implementation Details The Giraffa metadata server is implemented as a "coprocessor" embedded in each HBase region server [Fou], which works like a "stored procedure" in a relational database [RG00]. The current implementation of Giraffa relies on the underlying HBase to dynamically partition and distribute the metadata table across all its region servers to achieve load balancing. By default, HBase horizontally partitions its table as regions according to the size of existing regions. Since HBase is unaware of any semantic meaning of stored table contents, it will not deliberately partition a large directory or cluster small directories as IndexFS does. The default split threshold for an HBase region is as large as 10GB, which is much larger than the split threshold for directories in IndexFS. During the experiments, it was found Giraffa can easily suffer a skewed distribution in its lexicographic key space, in part due to its default schema for generating row keys. In order to mitigate this problem, we modified Giraffa's code by prepending a hash of the parent path to the original row key, and pre-split the namespace table in HBase at the beginning of each experiment. HBase allows users to pre-split tables to help better balance the system during the initial workload, provided that the key distribution is known beforehand. This trick allows Giraffa to immediately distribute the key space to all

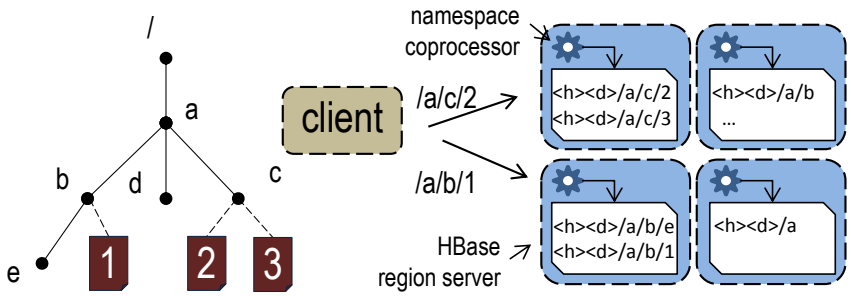


Figure 6.4: Giraffa stores its metadata in HBase, which partitions its table as a B-Tree. Each file or directory is mapped to an unique row at one HBase region server. The current implementation of Giraffa does not have hierarchical permission checking so no pathname resolution is performed.

region servers as soon as the system starts up in the experiments, achieving static load balance without the overhead of incremental data migration.

Relaxed Operation Semantics Giraffa relaxes semantics of several metadata operations. For access control, Giraffa does not check the permission information of every ancestor directory when accessing a file. This reduces the number of client-sever communications and helps with performance. To support the POSIX access control model, it could either adopt the metadata caching technique used by IndexFS, or use a schema like Lazy Hybrid (LH) [BMLX03] or CalvinFS that replicates a directory's permission bits to all files nested beneath it. The row key schema used by Giraffa also affects directory rename operations. Since the directory name is part of its children's row keys, rename requires read-modify-write on all of its child files. Because of the difficulty of supporting atomic rename of directories, it only supports rename of files in the same directory.

Fault Tolerance Giraffa servers translate file system operations into HBase operations and use a global lock to control concurrent operations. Giraffa server logic is embedded in the HBase region server as a coprocessor, and only keeps minimal state in memory, so that the Giraffa server shares its fate with its region server. The reliability and consistency of Giraffa are entirely ensured by the underlying HBase. HBase uses similar techniques as IndexFS for fault tolerance, including distributed write-ahead logging for failure recovery and ZooKeeper for membership monitoring.

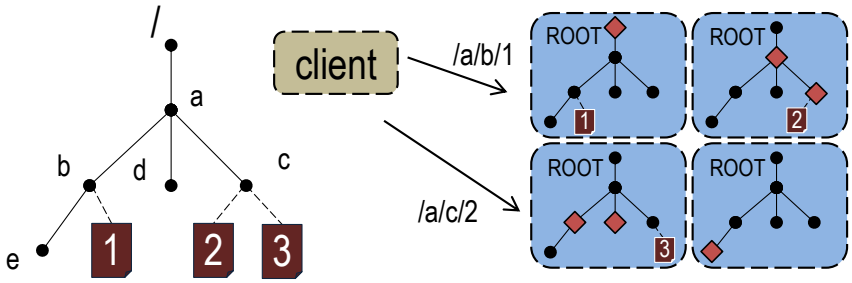


Figure 6.5: ShardFS replicates directory lookup state to all metadata servers so every server can perform path resolution locally. File metadata and non-replicated directory metadata is stored at exactly one server determined by a hash function on the full pathname.

6.2.2 Replicated directories with sharded files (ShardFS)

ShardFS scales distributed file system metadata performance by fully replicating directory lookup state across servers. Directory lookup state is the information required by accessing to an object specified by a pathname, which includes permission to lookup that object's name in its parent directory, and permission to lookup that parent directory's name in the grandparent directory. The policy is recursive as specified by the pathname back to either the file system's root or a directory currently open in the caller's process. The goal of replicating this information in ShardFS is to ensure that each file operation on a server is single-site [SMA+07] and avoids obtaining locks from multiple servers. This strategy slows down mutations that affect this information (changes to directory names, directory access permissions, or parent-child structure in the namespace) in order to speed up and load balance accesses to the objects reached by a successful pathname lookup as shown in Figure 6.5. Every metadata server contains a complete replica of this namespace information, so a single RPC to the appropriate metadata server will be sufficient to complete pathname resolution without additional RPCs. Unlike directory lookup states, file metadata is stored only in one metadata server. They are distributed by a sharding function (some hash function) using the pathname as input. The metadata server to which a pathname shards is defined as the primary metadata server for this pathname. When a pathname represents a file, its metadata is stored only on the primary metadata server.

With fully replicated directory lookup states in ShardFS, pessimistic multi-server locking for all metadata operations related to directories is normally required. As well, operations accessing metadata on one metadata server only are single-site transactions and can execute with one RPC to the primary metadata server (all metadata servers are internally serializable and transactional for metadata operations). Almost all operations on file metadata or non-replicated directory metadata, such as timestamps, are single RPC operations. The scaling benefit that these operations get from namespace replication is the primary motivation for ShardFS's design. Other operations are distributed transactions and use a two phase locking protocol, limiting overall scalability. This is the major difference between ShardFS and IndexFS.

File System Specific Optimizations

ShardFS strives to reduce the latency for single RPC operations on files by not blocking them on locks taken by other concurrent replicated state mutations. Specifically, ShardFS uses optimistic concurrency control for single RPC operation in as many transaction classes as possible and will fall back to retry with pessimistic two phase locking concurrency control when optimistic verification fails. ShardFS limits the number and semantics of file system metadata operations, and does not directly examine the entire read and write sets for concurrent operations as in traditional optimistic concurrency control. Instead, file system operation semantics are used to detect optimistic verification failures that cause single site transactions to abort and retry with full pessimistic locking.

The file system operations that are not single RPC can be classified into the following three classes:

- All operations that grant permissions to replicated states: create a directory (`mkdir`), permission operations (`chmod` +mode, or `chgrp` +group);
- All operations that deny permissions to replicated states: remove a directory (`rmdir`), permission operations (`chmod` -mode, or `chgrp` -group); and
- All operations that make non-monotonic changes on permissions of replicated state: for example, rename, mode changes that grant and deny permissions, changes to ownership.

For the first two classes, only monotonic changes to replicated states are performed. ShardFS makes the transaction protocol to be optimistic such that the transaction does not acquire pessimistic locking on each server. Instead, it resorts to a single RPC operations to detect inconsistent replicated states itself. If detecting

any inconsistency, the calling client then recognizes the conflict and retries the single RPC operation with pessimistic locking. For the third class of operations in ShardFS, the distributed transactions have to be maximally defensive – they cause ShardFS to serialize all concurrent operations with conflicting scope (even optimistic single RPC operations) by taking pessimistic locks at every server. This design choice made by ShardFS assumes that operations of the third class are rare in the target workloads. More details about its distributed transaction protocol can be found in the original paper [Xia13].

Readdir and Small Directories As specified in POSIX standards, `readdir` only requires returning directory entries. Directory entries are sharded for files and are replicated for subdirectories. ShardFS sends `readdir` to all metadata servers to gather every entry in the directory in parallel. The results returned from metadata servers will be consolidated by the ShardFS client. If `readdir` fails on any metadata server, the whole operation fails. Subdirectories are included in the `readdir` result as long as it exists in one metadata server. When a subdirectory is not present in all `readdir` results, it is either being created (and will succeed) or deleted. In both cases, it is reasonable to return the subdirectory to the client.

By default, ShardFS does not guarantee consistency for `readdir` with other concurrent operations. For example, `readdir` may or may not return a file created concurrently in the same directory during the completion of `readdir`. This is undefined in POSIX semantics, so the file system users are expected to handle both cases.

For small directories, many metadata servers don't contain any directory entry. `readdir` sent to these metadata servers won't return any data. This also implies that the directory metadata shouldn't be replicated to all metadata servers since they won't be used at all. ShardFS is not designed for small directories and small directories incur high resource overhead. It gets even worse when these directories are transient. One frequent pattern in Hadoop workloads is that many Hadoop jobs create a temporary directory for each task for isolation and replication. When the task completes, its output is renamed from the temporary directory to the final output directory. To mitigate this problem, ShardFS authors suggest avoiding the usage of temporary directory in Hadoop framework by adding prefixes to these output files. By doing so, creating all temporary directories can be effectively avoided and many directory operations can be changed to file operations without compromising the correctness of the Hadoop framework.

Implementation ShardFS is implemented as a new HDFS client library written in JAVA on top of standalone IndexFS servers [RG13, RZPG14] with modifications

to support server side pathname resolution. Sharing the same IndexFS code base provides a fair comparison between these two approaches. The namespace replication with optimized directory lookup state mutation operations is implemented on the client side. A lock server is implemented to resolve races among all directory lookup state mutation operations. Locking a pathname is equivalent to acquiring a write lock on the last component of the path and read locks on all ancestors.

Fault Tolerance ShardFS relies on high availability of its underlying metadata servers. IndexFS tolerates failures by replicating data in its underlying storage as described in previous sections. The lock server also tracks logs of outstanding distributed directory transactions to tolerate client failures. When a client fails, later lock acquisition on the same path will trigger a recovery process for the operation. A backup lock server or quorum system can be used to tolerate lock server failure. The current version of lock server does not implement fault tolerance strategy.

6.2.3 Comparison Summary

Table 6.2 summarizes the design difference among the three metadata systems. To analyze the performance implication of these designs under various workloads, We discuss a few major differences of the three systems:

RPC amplification of metadata operations RPC amplification is defined as the the number of RPCs sent for an operation. One major source of RPC amplifications comes from path resolution. POSIX semantics require accessing each ancestor directory on the path to check permissions. The path components may be stored on different servers with a distributed namespace, which require multiple RPCs to fetch. ShardFS replicates directory metadata to every metadata server for server local pathname resolutions, complicating the protocol of directory metadata mutation operations. In IndexFS, both clients and servers maintain a consistent cache of path attributes to reduce RPCs, which can be limited by cache effectiveness. Giraffa abandons the access control model of POSIX by only checking permission bits of the final component. By using the full pathname as part of the primary key, most single file metadata operations in Giraffa require only one RPC.

Another source of RPC amplifications is the partitioning and replication of metadata. For directory mutation metadata operations, the ShardFS client contacts all metadata servers to execute the distributed transactions, adding to RPC amplification. For IndexFS, attribute modification operations such as *chmod* and *chown* require only one RPC but may need to wait for lease expiry. Moreover, *mkdir* and

	Replicated directories with sharded files	**Dynamically partitioned namespace**	**Table partitioned namespace**
Example system	ShardFS	IndexFS	Giraffa
Metadata distribution	Replicated directory lookup states; sharded files	Partitioned into directory subsets	Partitioned by HBase
Metadata addressing	hash(pathname)	parent directory's inode num + hash(filename)	hash(path prefix) + depth + pathname
File operation (`stat`, `chmod`, `chown`, etc.)	1 RPC for all operations	Many RPCs for path traversal depending on cache locality	1 RPC for `stat` and chmod, 2 RPCs for mknod
Directory metadata mutations (`mkdir`, `chmod`, etc.)	Optimized distribution for monotonic operations	1 RPC but waiting for lease expiration	2 RPCs, similar to file operations
Concurrency control	Optimistic locking on the central server	Locking at directory partition level	Serialized by each tablet server
Client caching	Only cache server configuration	Cache path prefix ACLs and directory partition location	Cache the location of tablets
Load balancing	Load balanced file access by static hashing	Dynamically assign directory; split large directory by size	Pre-split and dynamically split tablets by size

Table 6.2: Summary of design choices made by three metadata services.

splitting involve two servers to perform transactions. And *rmdir* checks each partition to see if the directory is empty. The RPC amplification for *readdir* is proportional to the number of partitions and the overall size of each partition.

Metadata operation latencies Both Giraffa and ShardFS try to keep the latency of file metadata operations low as one round-trip RPC. In this case, latency is mostly affected by server load. For most directory metadata mutations in ShardFS, multiple RPCs are issued to all metadata servers in parallel. Thus its latency is sensitive to the slowest RPC. For IndexFS, the latency of metadata operations is affected by the hit ratio of the directory entry cache. Directory metadata mutations such as *chmod* and *chown* are also sensitive to the lease duration in IndexFS. Giraffa will have problems similar to IndexFS if it someday supports POSIX semantics for access control.

Consistency model for metadata operations All three metadata systems guarantee serializability for file metadata operations. The accuracy of access and modification time stamps for directories are relaxed for better performance in all three systems. *readdir* behavior under concurrency is not well defined in POSIX, which makes it flexible for system designers. All three systems provide an isolation level called "read committed" [BBG+95]. A *readdir* will always reflect a view of the directory at least as new as the beginning of the operation. It reflects all mutations committed prior to the operation, but may or may not reflect any mutation committed after the issue of the operation. This allows the systems to implement *readdir* as multiple independent requests to each directory partition without locking the entire directory.

Load balanced across metadata servers An important scalability factor is the balance of load across metadata servers. Overall performance is limited by the slowest server. IndexFS uses a dynamic growth policy such that a newly created directory starts with one server and is dynamically split as it grows. While this policy maintains locality for small directories, it may experience higher load variance when a directory is close to splitting. Servers containing the top of the namespace may get higher load due to client cache misses or renewal. Giraffa will experience similar problems as it also splits the tablet according to the size. Since the split is handled by HBase, a directory in Giraffa is not necessarily clustered nor well balanced across severs. ShardFS, in contrast, maintains both capacity and load balance by sharding files and replicating directory metadata at all times. It has a more balanced load across servers, but its directory metadata mutation is slower. Currently none of the systems implement strategies for the case where a few popular files dominate a workload [FLAK11].

Memory Resource Consumption One important hardware resource that will greatly affect the performance and scalability is memory, since all systems use large server caches to avoid load imbalance and disk accesses. The first type of cache is the directory entry cache that stores information such as inode number and permission bits that are necessary for path resolution and metadata addressing. The second type of cache is the memory index used to index the location of metadata on the secondary storage. The memory consumption of these caches is largely determined by the metadata distribution, the key schema and internal data structure used for indexing metadata entry. Giraffa does not need the directory entry cache because it uses full pathname as the key and its different access control model. ShardFS uses the trie data structure for the directory entry cache. IndexFS uses a hash table for the directory entry cache, but each IndexFS client also has a directory entry cache which results larger total memory consumption than Giraffa and ShardFS.

Scalability Both IndexFS and Giraffa scale in throughput as more servers are added to the system for most workloads. In contrast, ShardFS's directory metadata mutation operations get slower as servers are added due to replication. In the extreme, if the workload only contains directory metadata mutation operation, ShardFS with more servers gets slower. However, when the ratio between file and directory metadata mutation operation scales as the number of servers, ShardFS performance also scales. As the namespace grows larger, both systems face the challenge of maintaining an effective directory entry cache to avoid unnecessary RPCs and disk requests.

6.3 Experimental Evaluation

The evaluation section focuses on validating IndexFS's design choices and comparing its performance with other distributed solutions. This section tries to evaluate the following aspects:

- The scalability of distributing large directories and namespace;
- The effectiveness of caching for pathname resolution;
- The performance of load balancing in IndexFS compared to other solutions;
- The performance analysis of bulk insertions in IndexFS;

	Kodiak	Susitna	LANL Smog
#Machines	128	5	32
HW year	2005	2012	2010
OS	Ubuntu 12.10	CentOS 6.3	Cray Linux
Kernel	3.6.6 x86_64	2.6.32 x86_64	
CPU	AMD Opteron 252, 2-core 2.6 GHz	AMD Opteron 6272, 64-core 2.1 GHz	AMD Opteron 6136, 16-core 2.4 GHz
Memory	8GB	128GB	32GB
Network	1GE NIC	40GE NIC	Torus 3D 4.7GB/s
Storage	Western Digital 1TB disk/node	PanFS 5-shelf 5 MDS,50 OSD	HW RAID array 8GB/s bandwdith
Tested FS	HDFS, PVFS	PanFS	Lustre

Table 6.3: Three clusters used for experiments.

- The portability of IndexFS as middleware when layering on top of different existing distributed file systems.

The prototype of IndexFS is implemented in about 10,000 lines of C++ code using a modular design that is easily layered on existing cluster file systems such as HDFS [HDF], Lustre [Lus], PVFS [RL06], and PanFS [WUA+08]. Our current version implements the most common POSIX file system operations except `hardlink` and `xattr` operations. Some failure recovery mechanisms, such as replaying write-ahead logs, are not implemented yet.

All experiments are performed on one of three clusters. Table 6.3 describes the hardware and software configurations of the three clusters. The first cluster is a 128-node cluster taken from the 1000-node PRObE Kodiak cluster [GGJL]. It is used to evaluate IndexFS's scaling performance and design trade-offs. In this cluster, IndexFS is layered on top of PVFS or HDFS, and its performance is compared against ShardFS, Giraffa, PVFS and HDFS. The second cluster (PRObE Susitna [GGJL]) and the third cluster (LANL Smog [LGH+11]) are used to evaluate IndexFS's portability to PanFS and Lustre respectively. In all experiments, clients and servers are distributed over the same machines or partitioned into two groups. The client uses an IndexFS library API, and the threshold for splitting a partition is always 2,000 entries. In asynchronous commit mode, the IndexFS server flushes its write ahead log every 5 seconds or every 16KB (similar to Linux local file systems like Ext4 and

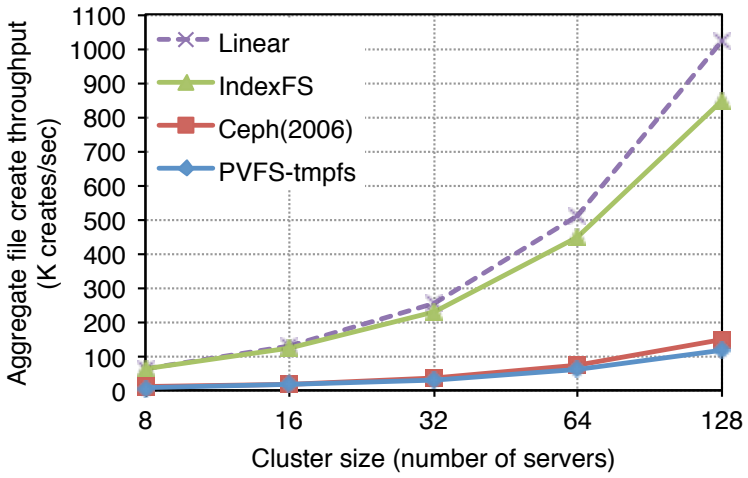


Figure 6.6: IndexFS on 128 servers deliver a peak throughput of roughly 842,000 file creates per second. The prototype RPC package (Thrift [thr]) limits its linear scalability.

XFS [MCB07, Swe96]). All tests were run for at least three times and the coefficient of variation of results is less than 2%.

6.3.1 Large Directory Scaling

This section shows how IndexFS scales to support large directories over multiple Kodiak servers. To understand its dynamic partitioning behavior, we start with a synthetic *mdtest* benchmark [mdt] to insert zero-byte files into a single shared directory [WBML06, PG11]. A three-phase workload has been generated. The first phase is a concurrent create workload in which eight client processes on each node simultaneously create files in a common directory. The number of files created is proportional to the number of nodes: each node creates 1 million files, so 128 million files are created on 128 nodes. The second phase performs `stat` on random files in this large directory. Each client process performs 125,000 `stat` calls. The third phase deletes all files in this directory in a random order.

Figure 6.6 plots aggregated operation throughput, in file creates per second, averaged over the first phase of the benchmark as a function of the number of servers (1 server and 8 client processes per node). IndexFS with SSTables and write-ahead logs stored in PVFS scales linearly up to 128 servers. IndexFS in this experiment

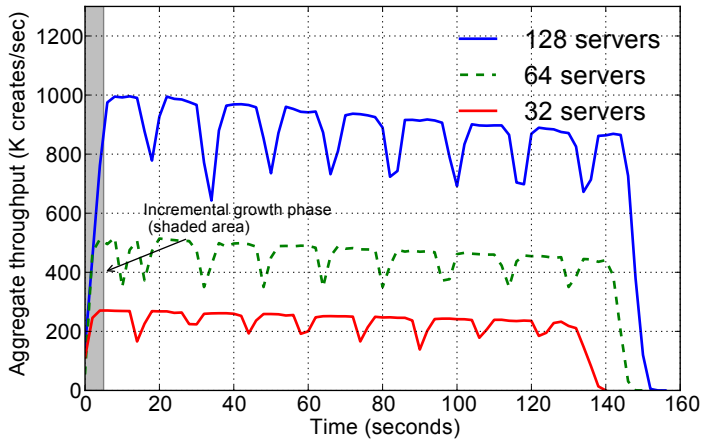


Figure 6.7: IndexFS achieves steady throughput after distributing one directory hash range to each available server. After scale-out, throughput variation is caused by the compaction process in LevelDB. Peak throughput degrades over time because the total size of the metadata table is growing, so negative lookups do more disk accesses.

uses only one LevelDB table to store metadata (without using column-style storage schema). With 128 servers, IndexFS can sustain a peak throughput of about 842,000 file creates per second, two orders of magnitude faster than current single server solutions.

Figure 6.6 also compares IndexFS with the scalability of Ceph and PVFS. PVFS is measured in the same Kodiak cluster, but since PVFS's metadata servers uses a transactional database (BerkeleyDB) for durability, which is stronger than IndexFS or Ceph, it is configured to store its records in a RAM disk to achieve better performance. When layered on top of Ext3 with hard disks, 128 PVFS servers only achieve one hundred creates per second. For Ceph, Figure 6.6 reuses numbers from the original paper [WBML06] [1]. Their experiments were performed on a cluster with a similar hardware and configuration. The reason that IndexFS outperforms other file systems is largely due to the use of log structured metadata layout.

Figure 6.7 shows the instantaneous creation throughput during the concurrent create workload. IndexFS delivers peak performance after the directory has become

[1]The directory splitting function in the latest version of Ceph is not stable. According to Ceph developers, the dynamic splitting function of current version of Ceph is often disabled when testing multiple metadata servers.

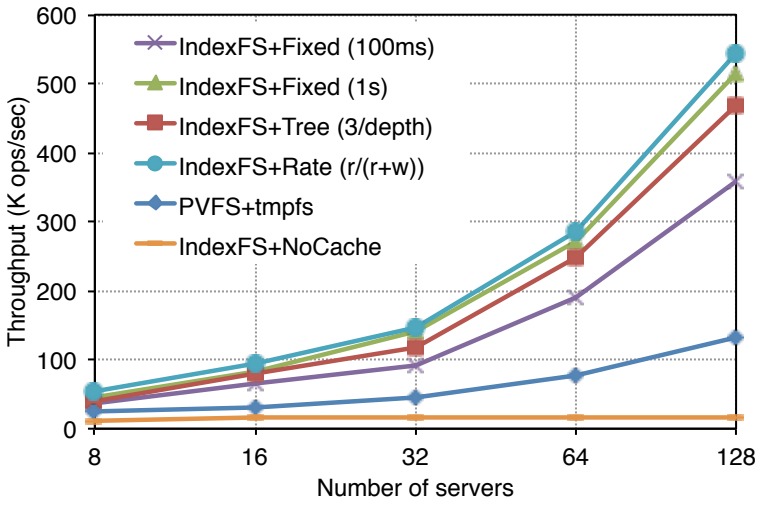


Figure 6.8: Average aggregate throughput of replaying 1 million operations per metadata server on different number of nodes using a one-day trace from a LinkedIn HDFS cluster.

large enough to be striped on all servers according to the GIGA+ splitting policy. During the steady state, throughput slowly drops as LevelDB builds a larger metadata store. This is because when there are more entries already existing in LevelDB, performing a negative lookup before each create has to search more SSTables on disk. The variation of the throughput during the steady state is caused by the compaction procedure in LevelDB.

IndexFS also demonstrates scalable performance for the concurrent lookup workload, delivering a throughput of more than 1,161,000 file lookups per second for our 128 server configuration. Good lookup performance is expected because the first few lookups fetch the directory partitions from disk into the buffer cache and the disk is not used after that. Deletion throughput for 128 server nodes is about 930,000 operations per second.

6.3.2 Metadata Client Caching

To evaluate IndexFS's client-side metadata caching, We replay a workload trace that records metadata operations issued to the namenode of a LinkedIn HDFS cluster covering an entire 24-hour period. An HDFS trace is used because it is the largest dynamic trace available to us. This LinkedIn HDFS cluster consists of about 1000

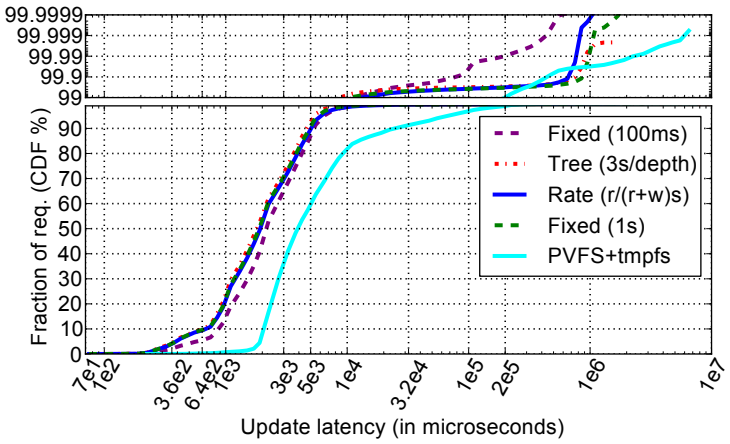
machines, and its namenode during the trace accessed about 1.9 million directories and 11.4 million files. The trace captures 145 million metadata operations of which, 84% are lookup operations (e.g., `open` and `getattr`), 9% are create operations (including `create` and `mkdir`), and the rest (7%) are update operations (e.g., `chmod`, `delete` and `rename`). Because HDFS metadata operations do not use relative addressing, each will do full pathname translation, making this trace pessimistic for IndexFS and other POSIX-like file systems.

Based on this trace, a two-phase workload is created. The first phase is to re-create the file system namespace based on the pathnames referenced in the trace. Since this benchmark focuses on metadata operations, all created files have no data contents. The file system namespace is re-created by multiple clients in parallel in a depth-first order. In the second phase, the first 128 million metadata operations recorded in the original trace are replayed against the tested system. During the second phase, eight client processes are running on each node to replay the trace concurrently. The trace is divided into blocks of subsequent operations, in which each block consists of 200 metadata operations. These trace blocks are assigned to the replay clients in a round-robin, time-ordered fashion. The replay phase is a metadata read intensive workload that stresses load balancing and per-query metadata read performance of the tested systems. IndexFS in this section uses the LevelDB-only metadata schema.

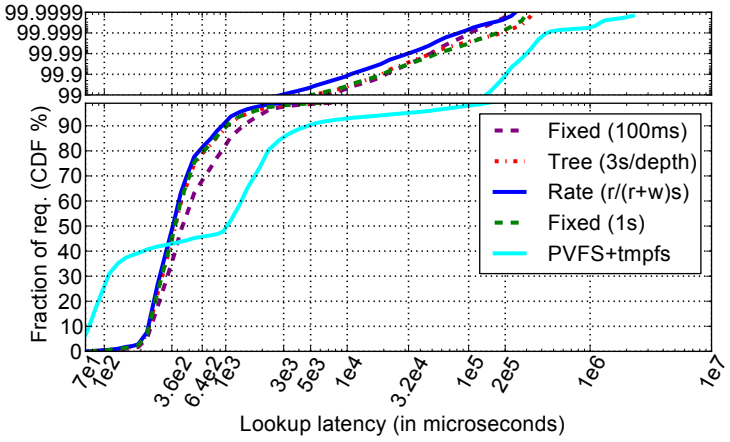
Figure 6.8 shows the aggregated throughput of the tested system averaged over the replay phase at different cluster scales ranging from 8 servers to 128 servers. In this experiment, IndexFS is compared with three client cache policies for the duration of directory entry leases: fixed duration (100 milliseconds and 1 second), tree-based duration ($3/depth$ seconds), and rate-based hybrid duration ($\frac{r}{w+r}$ seconds). The duration of all rate-based leases and most tree-based leases are shorter than 1 second. We also compare against IndexFS without directory entry caching and PVFS on tmpfs.

From Figure 6.8, we can see that IndexFS performance does not scale without client-based directory entry caching because performance is bottlenecked by servers that hold hot directory entries. Equipped with client caches of directory entries, all tested systems scale better, and IndexFS with rate-based caching achieves the highest aggregate throughput of more than 514,000 operations per second; that is, about 4,016 operations per second per server.

The reason that the aggregate throughput of rate based caching is higher than the other policy is because it provides more accurate predictions for the lease duration. Since this workload is metadata read intensive, longer average lease duration



(a) Update Latency



(b) Lookup Latency

Figure 6.9: Latency distribution of update operations (a) and lookup operations (b) under different caching policies ($6.4e2$ means 6.4×10^2). Rate-based policies offer the best average and 99% latency, which yields higher aggregate throughput.

can effectively reduce the number of unnecessary lookup RPCs between client and servers. So fixed duration caching with 1 second leases has higher average throughput than 100 millisecond leases. When increasing fixed duration lease to be 2 seconds and 4 seconds (not shown in the figure), the average throughput actually decreases because the latency delay of mutation operations now becomes more significant. In

comparison, the rate-based caching provides similiar average latency as 1 second fixed duration lease but has better control over the tail latency of mutation operations.

Figure 6.9 plots the latency distribution of lookup operations (e.g., `getattr`), and update operations (e.g., `chmod`) in the 128-node test. We can see that the rate-based case has the lowest median latencies and better 99th percentile latencies than all other policies. Its maximum write latency is higher than that of a fixed 100ms duration policy, because the rate based policy poorly predicts write frequencies of a few directory entries. PVFS has better 40th percentile lookup latency versus IndexFS because PVFS clients cache file attributes, but IndexFS clients do not; they cache name and permissions only. For `getattr` operation, IndexFS clients need at least one RPC, while the PVFS client may directly find all attributes in its local cache.

6.3.3 Load Balancing

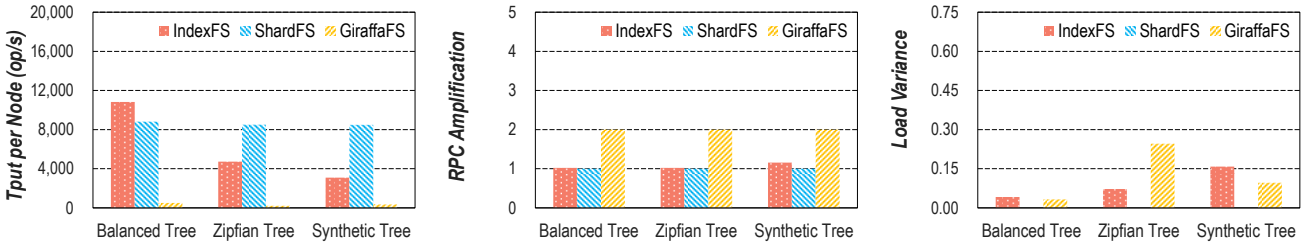
In this section, a set of microbenchmarks are executed on the three targeted file systems (IndexFS, ShardFS and Giraffa) to study the tradeoffs among their designs, especially on load balancing. Since the namespace structures also affect the load distribution, three distinct file system images are prepared to represent different namespace tree structures:

#**1. Balanced Tree**: In a balanced tree, each internal directory has 10 sub-directories, and each leaf directory has 1,280 children files. The height of the tree is 5. In total there are 111 K directories and 128 M files.

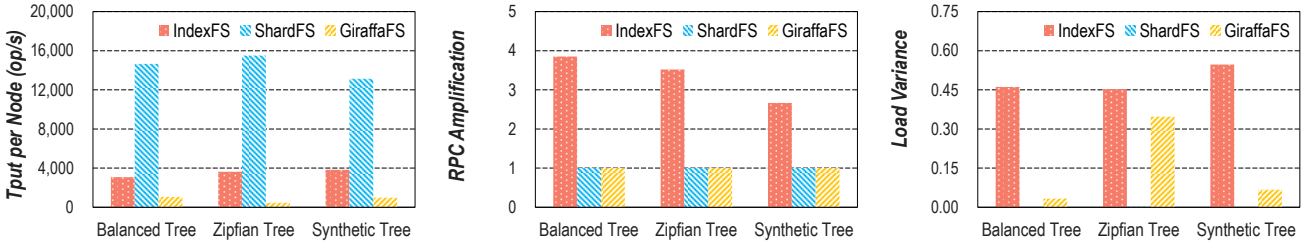
#**2. Zipfian Tree**: A Zipfian tree shares the same internal tree structure as a balanced tree, except that the sizes of its leaf directories are randomly generated according to a Zipfian distribution with an exponent parameter of 1.8. There are 111 K directories and approximately 128 M files inside the tree.

#**3. Synthetic Tree**: This tree is generated by a workload generator named Mimesis [ALR+12]. Mimesis models existing HDFS namespace images based on several statistical characteristics, such as the distribution of the number of files per directory and the number of sub-directories per parent directory. For this tree, Mimesis is used to model and scale an HDFS trace extracted from a Yahoo! cluster. The original Yahoo! namespace had 757 K directories and 49 M files. Mimesis expanded this file system image to contain 1.9 M directories and 128 M files with the same statistical characteristics as the original Yahoo! namespace.

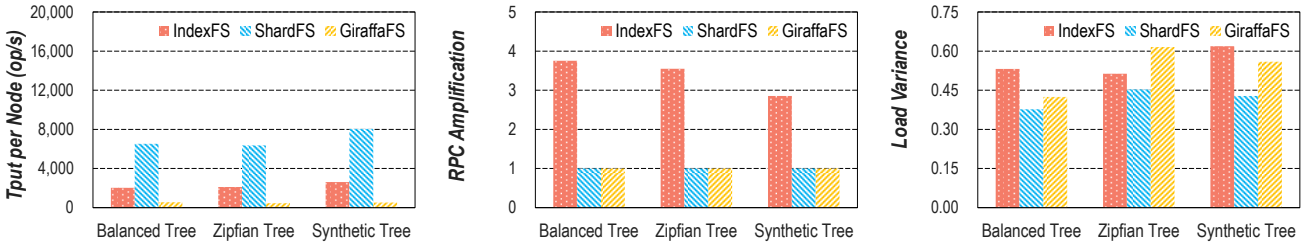
Similar to benchmarks in previous sections, a three-phase workload is used here. The first phase creates all internal directories. The second phase populates the



(a) Creation Phase: Throughput, RPC amplification, and Load variance



(b) Query Phase (with uniformly distributed file stats): Throughput, RPC amplification, and Load variance



(c) Query Phase (with Zipfian distributed file stats): Throughput, RPC amplification, and Load variance

Figure 6.10: Performance comparison among IndexFS, ShardFS, and Giraffa creating and stating zero-byte files with 64 server machines and 64 client machines.

namespace with empty files. During the third phase, each client performs *stat* on files randomly selected from the namespace. To model different access patterns, files to be accessed are chosen either uniformly or following a Zipfian distribution with an exponent parameter of 1.8. In all cases, the number of client threads is selected to saturate the target file system at its maximal throughput (an external tuning knob for our benchmark system). We measured three different metrics: *average throughput per server*, *RPC amplification*, and *load variance*. RPC amplification is reported as the total number RPCs over the total number of application-level file system operations. Load variance is measured as the coefficient of variation of the number of RPC requests received by each metadata server. All microbenchmark

experiments were run with 128 machines with 64 configured as servers and 64 as clients.

Figure 6.10 shows the experimental results for the file creation and query phases. In general, Giraffa appears to be much slower than both IndexFS and ShardFS. According to the profiling results, we believe the main reason lies in the less optimized code in Giraffa's implementation, such as inefficient memory copies, communicating overhead with HBase, as well as the use of global locks. As a result, in this section, we will mainly focus on RPC amplification and load variance when comparing with Giraffa.

In the file creation phase, IndexFS achieves its highest throughput in the balanced tree workload, since the other two workloads have a few very large directories. This gives rise to a set of hot servers performing necessary background activities to spread those large directories to multiple servers and balance the system for future operations. IndexFS also shows higher load variance in the later two workloads. This is because populating files for the Zipfian and synthetic trees produces namespace lookup requests that are imbalanced by nature. Fortunately, as files are created with depth-first order preserving path locality, the RPC amplification of IndexFS during the entire file creation phase is relatively low compared to that observed in the query phase. Unlike IndexFS, the performance of ShardFS tends to be stable for all three workloads. In fact, with files uniformly distributed across all of its servers, ShardFS can often achieve good load balance for file creates regardless of the actual file system tree structure. Different from both IndexFS and ShardFS, a large directory in Giraffa can easily be held entirely by a single HBase region server and become a performance bottleneck. In addition to this vulnerability to load imbalance, Giraffa also shows higher RPC amplification. As Giraffa has to check the existence of the parent directory when creating a new file, there is an additional RPC for almost all file creation operations, This is because parent directories are very likely to be distributed to a remote region server according to Giraffa's current namespace partitioning strategy. In fact, according to POSIX semantics, Giraffa should have consulted all ancestor directories before it can ever create a new file. Unfortunately, if enforced, this can only lead to even more severe RPC overhead.

For the query phase with uniform file selection, ShardFS shows excellent load balancing, lower RPC amplification, and higher throughput. This is because ShardFS always distributes files evenly across all of its metadata servers and each metadata server can perform pathname lookups locally without contacting peer servers. However, for ShardFS, pathname lookup is not free in terms of CPU consumption at the server side. When files are located deeper in the namespace, ShardFS has to pay more

CPU cycles to find parents and check permissions, which can, to some extent, lower its overall throughput. As is demonstrated in the synthetic tree workload, ShardFS's throughput is 14% less than that observed in the Zipfian tree microbenchmark. Unlike ShardFS, IndexFS's performance is largely limited by its RPC amplification, which can in part be attributed to this random read access pattern. These requests make IndexFS's client-side lookup cache relatively useless, causing more cache misses and forcing IndexFS clients to frequently fetch lookup states from servers. However, since IndexFS is able to dynamically split large directories, it doesn't get bottlenecked on large directories even with a skewed namespace such as the Zipfian and synthetic trees. In fact, IndexFS performs better in these namespaces as its client-side lookup cache becomes more effective in the later stage of these workloads. This leads to lower RPC amplification, and higher throughput, albeit higher load variance. For Giraffa, due to it's inability to split large directories, its load variance is largely determined by the shape of the namespace tree. For example, it shows higher load variance when it comes to the Zipfian tree. In addition, since Giraffa does not actually perform pathname resolution like IndexFS and ShardFS; the RPC amplification for Giraffa is always one for all file stat operations. Finally, all the 3 file systems show lower performance and higher load variance when files are selected following the Zipfian distribution.

However, IndexFS should be able to gain certain performance benefits under such access pattern, as its clients are more likely to reuse a prefix cache entry before the entry expires. This helps reduce lookup RPCs and therefore can improve overall system efficiency. Unfortunately, the performance impact of the major compaction done in IndexFS's background process is also amplified under the Zipfian distribution, which results in lower performance. In fact, improved system throughput is observed when running the experiment for a prolonged period. In this case, the side effect of the major compaction is better amortized and the benefits of the aforementioned cache locality manifests more effectively.

In summary, IndexFS's file stat performance is mainly a function of its cache effectiveness. ShardFS is able to deliver deterministic fast file stat performance by replicating directory lookup state. Giraffa often suffers load imbalance even without performing pathname resolution.

6.3.4 Bulk Insertion and Factor Analysis

This experiment investigates four optimizations contributing to bulk insertion performance. The following configuration is used to break down the performance difference between base server-side execution and client-side bulk insertion:

- **IndexFS** is a base server-executed operation with synchronous write-ahead logging in the server;
- +**async** enables asynchronous write-ahead logging (4KB buffer) in the server, increasing the number of recent operation vulnerable to server failure; this is almost the configuration used in the experiments of Section 6.3 parts A through C, which flushes the write-ahead log every 5 seconds or 16KB.
- +**bulk** enables client-side bulk insertion to avoid RPC overhead with asynchronous client side write ahead logging;
- +**column-style** enables column-style storage schema in the client-side when the client builds SSTables; and
- +**larger buffer** uses a larger buffer (64KB) for write-ahead logging, increasing the number of recent operations vulnerable to server failures.

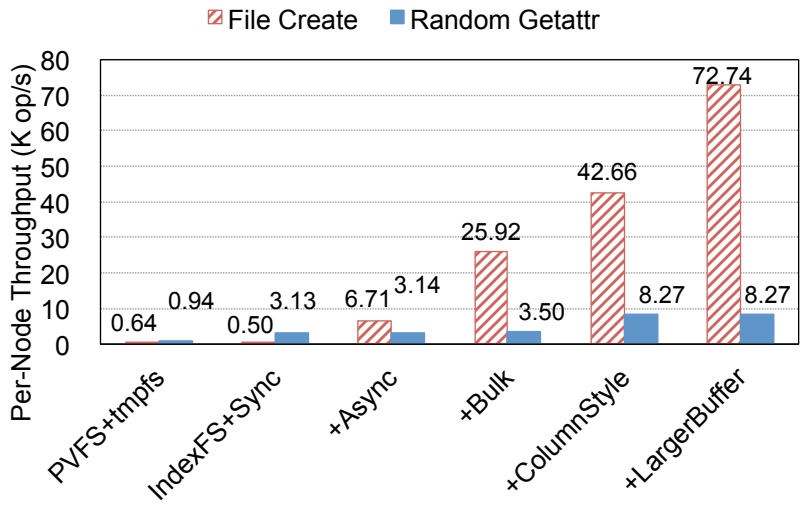


Figure 6.11: Contribution of optimizations to bulk insertion performance on top of PVFS. Optimizations are cumulative.

All experiments are run with 8 machines in the Kodiak cluster, each hosting 16 client processes and 1 IndexFS server process, a load high enough to benefit from group commits. The workload is the *mdtest* benchmark used in Section 6.3.1. We compare the performance of native PVFS (using tmpfs) with IndexFS layered on top of PVFS (using Ext3 on a disk). Figure 6.11 shows the performance results. In general, combining all optimizations improves file creation performance by 113 times compared to original PVFS mounted on tmpfs. Asynchronous write-ahead logging can bring 13 times improvement to file creation by buffering 4KB of updates before writing. Bulk insertion avoids overheads incurred by per-operation RPC to the server and compactions in the server. This brings another 3 times improvement. Using a column-style storage schema in the client helps with both file creation and lookup performance since the memory index caches well. The improvement to file creation speed provided by enlarging the write-head log buffer increases sub-linearly because it does not reduce the disk traffic caused by building and writing SSTables.

6.3.5 Portability to Multiple File Systems

To demonstrate the portability of IndexFS, we run the *mdtest* benchmark and *checkpoint* benchmarks [NB08] when layering IndexFS on top of three cluster file systems including HDFS, Lustre and PanFS. The experiment on HDFS is conducted on the Kodiak cluster with 128 nodes, the experiment on PanFS is conducted on the smaller Susitna cluster with 5 nodes, and the experiment on Lustre is on a third cluster at Los Alamos National Laboratory (Smog). The three clusters have different configurations, so a comparison between systems is not valid. The setup of the *mdtest* benchmark is similar to the one described in Section 6.3.1, and IndexFS uses the fixed 100ms duration metadata caching with LevelDB-only metadata schema and no client writeback caching.

Figure 6.12 shows the average per-server and aggregated throughput during the *mdtest* benchmarks when layering IndexFS on top of each of the three file systems, and is compared against the original underlying file systems. HDFS and Lustre only support one metadata server. PanFS supports a static partition of the namespace (each subtree at the root directory is a partition called a volume) over multiple metadata servers. Thus, we compare IndexFS to native PanFS by creating 1 million files in 5 different directories (volumes) owned by 5 independent metadata servers.

For all three configurations and all metadata operations, IndexFS has made substantial performance improvements over the underlying distributed file systems by reusing their scalable client accessible data paths for LSM storage of metadata. The

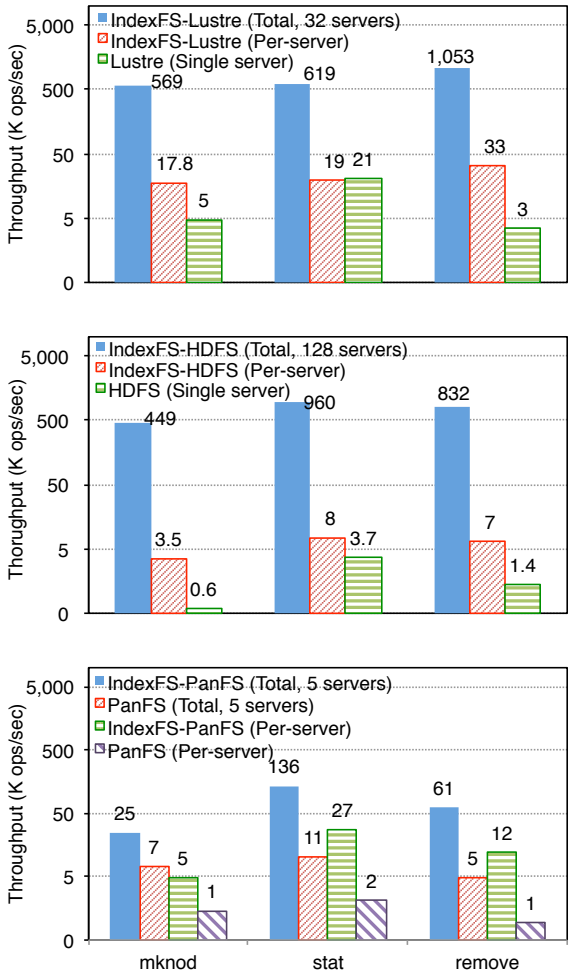


Figure 6.12: Per-server and aggregated throughput during mdtest with IndexFS layered on top of Lustre (on Smog), HDFS (on Kodiak), and PanFS (on Susitna) on a log scale. HDFS and Lustre have only one metadata server.

lookup throughput of IndexFS on top of PanFS is extremely fast because IndexFS packs metadata into file objects stored in PanFS, and PanFS has more aggressive data caching than HDFS. Compared to native Lustre, IndexFS's use of LSM tree improves file creation and deletion. However, for *stat*, it achieves only similar per-server performance because Lustre's clients also cache attributes of files created in the first phase.

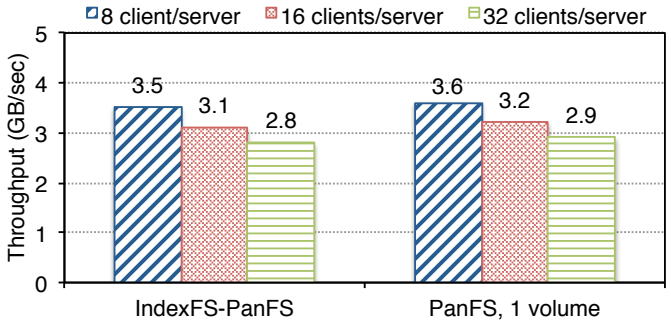


Figure 6.13: The aggregate write throughput for the N-N checkpointing workload. Each machine generates 640 GB of data.

We use Los Alamos National Lab's filesystem checkpoint benchmark [NB08] on Susitna and PanFS storage to test the overhead of IndexFS's middleware approach on the data-path bandwidth for large file reads and writes. In the checkpoint benchmark, N processes each independently write a single checkpoint file in the same directory; this is called "N-N checkpointing". All processes are synchronized using a barrier before and after writing the checkpoint file. In this test, we also vary the number of client processes per node from 8 to 32 clients. Each client process will generate a total of $640GB/\#clients$ amount of checkpoint data to the underlying file system. The size of the per-call data buffer is set to be 16KB. For IndexFS, the checkpoint files generated in the test will first store 64KB in the metadata table, and then migrate this 64KB and the rest of the file to the underlying distributed file system. Figure 6.13 shows the average throughput during the write phase in the N-N checkpoint workload. IndexFS's write throughput is comparable to the native PanFS, with an overhead of at most 3%. Reading these checkpoint files through IndexFS has a similar small performance overhead.

6.4 Summary of IndexFS Benefits

Many cluster file systems lack a general-purpose scalable metadata service that distributes both namespace and directories. IndexFS is built to allow *existing* file systems to deliver scalable and parallel metadata performance by reusing their original scalable data path. The experiments have demonstrated that IndexFS delivers a fifty percent to two orders of magnitude improvement in the metadata performance over several existing file systems including PVFS, HDFS, Lustre, and Panasas's PanFS.

Compared to other distributed solutions including ShardFS and Giraffa, IndexFS has better balanced performance for all types of metadata operations.

There are several major key ideas in IndexFS design. First, IndexFS adopts an efficient combination of scale-out indexing techniques with a scale-up metadata representation to enhance the scalability and performance of the metadata service. Secondly, since the pathname resolution is the performance bottleneck, client caching with minimal server state is used to enhance load balancing and insertion performance for creation-intense workloads. Finally, IndexFS uses a portable design that works with existing file system deployment with few configuration changes, and reuses their data path to provide fast access on the metadata path.

Chapter 7

Conclusion and Future Work

This dissertation proposed three novel systems to scale file system metadata management in both local and distributed file systems, whose design is informed by the underlying hardware as well as the workload analysis.

The dissertation explores the modular design of metadata management for local file system by layering the system on top of the key-value store engine and the object storage. We demonstrate that packing file and directory attributes into larger objects can effectively enhance locality and metadata performance. By further optimizing the in-memory index in terms of access performance and space efficiency, the local file system becomes more balanced and gains great performance enhancement on the solid-state disk. IndexFS provides a good example of combining scale-out indexing technique with local metadata store to build a distributed service. By identifying the bottleneck in the distributed system, storm-free metadata caching is used to eliminate hot spots. Bulk-insertion is also proposed to achieve higher creation throughput. With all these techniques, IndexFS can scale the metadata management of many existing file system.

In summary, this dissertation provided multiple sources of evidence to demonstrate that file system metadata management system can be built in an efficient and scalable way, which can meet future needs of exascale computing clusters. Finally, we believe many of lessons and techniques provided by this work will apply generally to all future storage systems.

7.1 Future Work

While this dissertation has shown promising scalability and performance for file system metadata management, there are several directions of further work that this dissertation leaves open.

Client-Funded Metadata Service: The design of IndexFS assumes the classic client-server model in a data center environment. However, metadata intensive workloads are still likely to bottleneck at the file system metadata servers due to namespace synchronization, which slows down application performance through lock contention on directories, transaction serialization, and RPC overheads. While IndexFS proposes a series of techniques to mitigate those synchronization overheads, the overall metadata performance of multiple dedicated metadata servers will be finally limited by the maximum metadata performance that this number of machines is able to deliver.

A bolder design extreme is to use a client-driven file system metadata architecture that allows applications to handle their own metadata operations locally in most of the time without any server intervention [ZRG14]. Unlike existing file systems that dedicate metadata server processes and machines to coordinate every metadata request in a centralized way, the file system can avoid inefficient RPC overheads and safeguards applications from unnecessary resource contention at the server side, effectively allowing the system to scale beyond a fixed sized control plane and utilize the resource available in the client sides. This envisions a stronger version of bulk insertion than the one used in IndexFS. Without centralized metadata server, this client-funded design faces challenges to validate the consistency of file systems after mutation: it requires "proof" of the correctness and authorization of these mutations bulk inserted by the clients. How to minimize the overhead of proofing remains an open problem.

Non-Volatile Memories: SlimFS is optimized for solid-state disks by increasing the granularity of in-memory index and spending more CPU cycles to compress the index. Non-volatile memories can be treated as storage devices instead of "persistent memory". However, when technologies like PCM and memristor can deliver nanosecond access times, their use as a DRAM replacement or substitute becomes more attractive. Recent works of evaluating database with different indexing structures on non-volatile memories [APD15] have begun to shed light on better ways to use these systems. It shows that some legacy components actually incur additional overheads when running on top of non-volatile memory, which suggests re-designing the storage systems for non-volatile memory is necessary.

Bibliography

[AADAD09] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Generating realistic impressions for file-system benchmarking. In *Proccedings of the 7th conference on File and Storage Technologies (FAST)*, 2009.

[ABC+02] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[ALR+12] Cristina L Abad, Huong Luu, Nathan Roberts, Kihwal Lee, Yi Lu, and Roy H Campbell. Metadata traces and workload models for evaluating big storage systems. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing (UCC)*. IEEE Computer Society, 2012.

[APD15] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. Let's talk about storage: Recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015.

[Arg95] Lars Arge. The buffer tree: A new technique for optimal i/o-algorithms. In *Proceedings of the Workshop on Algorithms and Data Structures*, pages 334–345. Springer-Verlag, 1995.

[AT15] Daniel J. Abadi Alexander Thomson. Calvinfs: Consistent wan replication and scalable metadata management for distributed file systems.

In *Proceedings of 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[AV88] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[azu] Windows azure storage-4 trillion objects and counting. http://blogs.msdn.com/b/windowsazure/archive/2012/07/18/windows-azure-storage-4-trillion-objects-and-counting.aspx.

[BBC+11] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, 2011.

[BBD09] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In *Proceedings of the 17th European Symposium on Algorithms (ESA)*, pages 682–693, 2009.

[BBG+95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *ACM SIGMOD Record*, volume 24, 1995.

[BG81] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Survey*, 13, 1981.

[BGG+09] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the ACM/IEEE conference on Supercomputing (SC)*, 2009.

[BGvK+06] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. Monetdb/xquery: A fast xquery processor powered by a relational engine. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2006.

[Blo70] B.H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communication of ACM 13, 7*, 1970.

[BM72] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1972.

[BMK10] Jack Burbank, David Mills, and William Kasch. Network time protocol version 4: Protocol and algorithms specification. *Network*, 2010.

[BMLX03] Scott A Brandt, Ethan L Miller, Darrell DE Long, and Lan Xue. Efficient metadata management in large distributed storage systems. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*. IEEE Computer Society, 2003.

[BS80] Jon Louis Bentley and James B Saxe. Decomposable Searching Problems I: Static to Dynamic Transformation. *Journal of Algorithms*, 1:301–358, 1980.

[CAK12] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive query processing in big data systems: A cross-industry study of MapReduce workloads. *PVLDB*, 5(12):1802–1813, 2012.

[CDE+12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[Cea96] Peter Corbett and et al. Overview of the mpi-io parallel i/o interface. In *Input/Output in Parallel and Distributed Computer Systems*, pages 127–146. Springer, 1996.

[CG86] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1:133–162, 1986.

[CKRT04] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 30–39. Society for Industrial and Applied Mathematics, 2004.

[CMKL+09] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, and S. Zdonik. A demonstration of scidb: A science-oriented dbms. *Proceedings of VLDB Endowment*, 2(2), 2009.

[Cus94] H. Custer. Inside the windows NT file system. *Microsoft Press*, 1994.

[Day08] Shobhit Dayal. Characterizing HEC storage systems at rest. In *Carnegie Mellon University, CMU-PDL-08-109*, 2008.

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

[DH06] John R. Douceur and Jon Howell. Distributed directory service in the farsite file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[ewm] Wikipedia: Exponential Moving Weighted Average. http://en.wikipedia.org/wiki/Moving_average.

[FAK13] Bin Fan, David G. Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[FAKM14] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (Co-Next)*, 2014.

[FBZ+14] Sorin Faibish, John Bent, Jingwang Zhang, Aaron Torres, Brett Kettering, Gary Grider, and David Bonnie. Improving small file performance with PLFS containers. Technical Report LA-UR-14-26385, Los Alamos National Laboratory, 2014.

[Fik] Andrew Fikes. Storage Architecture and Challenges (Jun 2010). Talk at the Google Faculty Summit 2010.

[FKP11] Nikolaos Fountoulakis, Megha Khosla, and Konstantinos Panagiotou. The multiple-orientability thresholds for random hypergraphs. In *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms (SODA)*, pages 1222–1236, 2011.

[FLAK11] Bin Fan, Hyeontaek Lim, David G Andersen, and Michael Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC)*, page 23. ACM, 2011.

[Fou] Apache Software Foundation. Hbase: the hadoop database, a distributed, scalable, big data store. http://hbase.apache.org/.

[FTXG11] Bin Fan, Wittawat Tantisiriroj, Lin Xiao, and Garth Gibson. Diskreduce: Replication as a prelude to erasure coding in data-intensive scalable computing, 2011.

[fus] FUSE. http://fuse.sourceforge.net/.

[FZL+13] Bin Fan, Dong Zhou, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. When cycles are cheap, some tables can be huge. In *Proceedings of the 14th USENIX conference on Hot Topics in Operating Systems (HotOS)*, 2013.

[GGJL] Garth Gibson, Gary Grider, Andree Jacobson, and Wyatt Lloyd. Probe: A thousand-node experimental cluster for computer systems research.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[Gir13] Giraffa: A distributed highly available file system. https://code.google.com/a/apache-extras.org/p/giraffa/, 2013.

[GK97] Gregory R. Ganger and M. Frans Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATC)*, 1997.

[GK10] Goetz Graefe and Harumi Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT)*, 2010.

[GNA+98] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. 1998.

[Hal05] Michael Austin Halcrow. eCryptfs: An Enterprise-class Encrypted Filesystem for Linux. *Proc. of the Linux Symposium, Ottawa, Canada*, 2005.

[HBD+14] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand S Aiyer, Liyin Tang, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Analysis of hdfs under hbase: a facebook messages case study. In *Proceedings of the 12th USENIX conference on file and storage technologies (FAST)*, pages 199–212, 2014.

[HDF] HDFS. Hadoop file system. http://hadoop.apache.org/.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference (ATC)*, volume 8, page 9, 2010.

[HLM94] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *USENIX Winter Technical Conference*, 1994.

[HMSC87] R. Haskin, Y. Malachi, W. Sawdon, and G. Chan. Recovery management in quicksilver. *In Proceedings of the Eleventh ACM Symposium on Operating System Principles (SOSP)*, 1987.

[Hyp13a] HyperLevelDB. A facebook fork of leveldb which is optimized for flash and big memory machines, 2013. https://rocksdb.org.

[Hyp13b] HyperLevelDB. A fork of leveldb intended to meet the needs of hyperdex while remaining compatible with leveldb, 2013. https://github.com/rescrv/HyperLevelDB.

[IKM07] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. In *Third Biennial Conference on Innovative Data Systems Research (CIDR)*, 2007.

[Jac88] Guy Joseph Jacobson. *Succinct Static Data Structures*. PhD thesis, Pittsburgh, PA, USA, 1988. AAI8918056.

[JBFL10] William K. Josephson, Lars A. Bongo, David Flynn, and Kai Li. DFS: a file system for virtualized flash storage. In *Proceedings of the 8th USENIX conference on file and storage technologies (FAST)*, 2010.

[Jea11] Stephanie Jones and et al. Easing the burdens of HPC file management. 2011.

[JNS+97] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and Rama Kanneganti. Incremental organization for data recording and warehousing. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)*, 1997.

[Kar09] Jan Kara. Ext4, BTRFS, and the others. In *Proceeding of Linux-Kongress and OpenSolaris Developer Conference*, 2009.

[Kas04] Aditya Kashyap. File system extensibility and reliability using an in-kernel database. *Master Thesis, Computer Science Department, Stony Brook University*, 2004.

[Kat97] Jeffrey Katcher. Postmark: A new file system benchmark. In *NetApp Technical Report TR3022*, 1997.

[LAK13] Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Practical batch-updatable external hashing with sorting. In *Proceedings of the 15th Meeting on Algorithm Engineering and Experiments (ALENEX)*, pages 173–182, 2013.

[LCL+09] Samuel Lang, Philip Carns, Robert Latham, Robert Ross, Kevin Harms, and William Allcock. I/O performance challenges at leadership scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2009.

[Lea09] Andrew Leung and et al. Magellan: A searchable metadata architecture for large-scale file systems. Technical Report UCSC-SSRC-09-07, University of California, Santa Cruz, 2009.

[Lev11] LevelDB. A fast and lightweight key/value database library, 2011. http://code.google.com/p/leveldb/.

[LFAK11] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: a memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[LGH+11] Cory Lueninghoener, Daryl Grunau, Timothy Harrington, Kathleen Kelly, and Quellyn Snead. Bringing up Cielo: experiences with a Cray XE6 system. In *Proceedings of the 25th international conference on Large Installation System Administration (LISA)*, 2011.

[LHAK14] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2014.

[LPG+17] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)*, 2017.

[LR04] Barbara Liskov and Rodrigo Rodrigues. Transactional file systems can be fast. *Proceedings of the 11th ACM SIGOPS European Workshop*, 2004.

[LSHC15] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[LSW14] Youyou Lu, Jiwu Shu, and Wei Wang. ReconFS: a reconstructable file system on flash storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[LSZ13] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the Lifetime of Flash-based Storage through Reducing Write Amplification from File Systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

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

[Mar12] Marcel Kornacker and Justin Erickson. Cloudera Impala: Real Time Queries in Apache Hadoop, For Real. http://blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real/, 2012.

[MB11] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *Proceedings of the 9th USENIX conference on File and Storage Technologies (FAST)*, 2011.

[MCB07] Avantika Mathur, Mingming Cao, and Suparna Bhattacharya. The new Ext4 filesystem: current status and future plans. In *Ottawa Linux Symposium*, 2007.

[mdt] mdtest: HPC benchmark for metadata performance. http://sourceforge.net/projects/mdtest/.

[Mea11] Micheal Moore and et al. OrangeFS: Advancing PVFS. *FAST Poster Session*, 2011.

[MG99] Marshall Kirk McKusick and Gregory R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. *Proceedings of the annual conference on USENIX Annual Technical Conference (ATC)*, 1999.

[Mit01] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1094–1104, 2001.

[MKM12] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*, 2012.

[MT84] Sape J. Mullender and Andrew S. Tanenbaum. Immediate files. *Software-Practice and Experience*, 1984.

[NB08] James Nunez and John Bent. LANL MPI-IO Test. http://institutes.lanl.gov/data/software/, 2008.

[New08] Henry Newman. HPCS Mission Partner File I/O Scenarios, Revision 3. http://wiki.lustre.org/images/5/5a/Newman_May_Lustre_Workshop.pdf, 2008.

[NVCF08] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. *ACM Transactions on Computer Systems, Vol.26, No.3 Article 6*, 2008.

[OCGO96] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 1996.

[Ols93] Michael A. Olson. The design and implementation of the Inversion file system. In *USENIX Winter Technical Conference*, 1993.

[OWZS13] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2013.

[PG07] Swapnil Patil and Garth Gibson. GIGA+: scalable directories for shared file systems. In *Proceedings of the 2nd workshop on parallel data storage (PDSW)*, 2007.

[PG11] Swapnil Patil and Garth Gibson. Scale and concurrency of GIGA+: File system directories with millions of files. In *Proceedings of 10th USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[RBM12] Ohad Rodeh, Josef Bacik, and Chris Mason. BRTFS: The Linux B-tree Filesystem. *IBM Research Report RJ10501 (ALM1207-004)*, 2012.

[RG00] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. Osborne/McGraw-Hill, 2000.

[RG13] Kai Ren and Garth Gibson. TableFS: Enhancing metadata efficiency in the local file system. *Usenix Annual Technical Conference (ATC)*, 2013.

[RKBH13] Kai Ren, YongChul Kwon, Magdalena Balazinska, and Bill Howe. Hadoop's adolescence: an analysis of hadoop usage in scientific workloads. *Proceeding of Very Large Database Endowment (PVLDB)*, 6(10):853–864, 2013.

[RL06] Robert Ross and Robert Latham. PVFS: a parallel file system. In *Proceedings of the ACM/IEEE conference on Supercomputing*, 2006.

[RO91] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, 1991.

[Rod08] Ohad Rodeh. B-trees, shadowing, and clones. *Transactions on Storage*, 2008.

[RZPG14] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. Scaling file system metadata performance with stateless caching and bulk insertion. In *Proceedings of the conference on high performance computing networking, storage and analysis (SC)*, 2014.

[s3t] Amazon s3-two trillion objects, 1.1 million requests/second. http://aws.typepad.com/aws/2013/04/amazon-s3-two-trillion-objects-11-million-requests-second.html.

[SAB+05] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam

Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-Store: A Column-oriented DBMS. In *Proceedings of the 31st international conference on Very large data bases (VLDB)*, 2005.

[SBMS93] Margo I. Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. *USENIX Winter Technical Conference*, 1993.

[SC05] Michael Stonebraker and Ugur Cetintemel. " one size fits all": an idea whose time has come and gone. In *Proceedings of 21st International Conference on Data Engineering (ICDE)*, 2005.

[Sch03] Philip Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, 2003.

[SCS+08] Adam Silberstein, Brian F. Cooper, Utkarsh Srivastava, Erik Vee, Ramana Yerneni, and Raghu Ramakrishnan. Efficient bulk insertion into a distributed ordered table. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 2008.

[Sel08] Margo Seltzer. Beyond relational databases. *Communications of the ACM*, 51(7), 2008.

[SGM+00] Margo Seltzer, Gregory Ganger, Kirk McKusick, Keith Smith, Craig Soules, and Christopher Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. *Proceedings of the annual conference on USENIX Annual Technical Conference (ATC)*, 2000.

[SH02] Frank B. Schmuck and Roger L. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[Shv10] Konstantin V Shvachko. Hdfs scalability: The limits to growth. *USENIX login*, 35:6–16, 2010.

[SIG07] Russell Sears, Catharine Van Ingen, and Jim Gray. To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem? *Microsoft Technical Report*, 2007.

[SKG+12] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project Voldemort. In *Proceedings of the 10th USENIX conference on file and storage technologies (FAST)*, 2012.

[SMA+07] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, 2007.

[SR12] Russell Sears and Raghu Ramakrishnan. bLSM: a general purpose log structured merge tree. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2012.

[SSM+13] Pradeep Shetty, Richard Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building workload-independent storage with VT-Trees. In *Proccedings of the 11th conference on File and Storage Technologies (FAST)*, 2013.

[Swe96] Adam Sweeney. Scalability in the XFS file system. In *Proceedings of the 1996 USENIX Annual Technical Conference (ATC)*, 1996.

[TB13] Aaron Torres and David Bonnie. Small file aggregation with PLFS. http://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-13-22024, 2013.

[thr] Apache thrift. http://thrift.apache.org.

[VSK+03] Murali Vilayannur, Anand Sivasubramaniam, Mahmut Kandemir, Rajeev Thakur, and Robert Ross. Discretionary caching for I/O on clusters. In *Proceedings of 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, 2003.

[WBML06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Darrell D. E. Long. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[Whe10] Ric Wheeler. One billion files: pushing scalability limits of linux filesystem. In *Linux Foundation Events*, 2010.

[WJea15] Yang Zhan William Jannen, Jun Yuan and et al. BetrFS: A Right-Optimized Write-Optimized File System. In *Proceedings of the 13th USENIX conference on file and storage technologies (FAST)*, 2015.

[WN13] Brent Welch and Geoffrey Noer. Optimizing a hybrid ssd/hdd hpc storage system based on file size distributions. *29th IEEE Conference on Massive Data Storage*, 2013.

[WSSZ07] Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Extending ACID Semantics to the File System. *ACM Transactions on Storage*, 2007.

[WUA+08] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX conference on File and Storage Technologies (FAST)*, 2008.

[WXSJ15] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data. In *USENIX Annual Technical Conference (ATC)*, 2015.

[Xia13] Lin Xiao. Scaling metadata service for weak scaling workloads. http://www.cs.cmu.edu/ lxiao/proposal.pdf, 2013.

[ZAADAD12] Yiying Zhang, Leo Prasath Arulraj, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. De-indirection for flash-based SSDs with nameless writes. In *Proceedings of the 10th USENIX conference on file and storage technologies (FAST)*, 2012.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 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 (NSDI)*. USENIX Association, 2012.

[ZFS] ZFS. http://www.opensolaris.org/os/community/zfs.

[ZG07] Zhihui Zhang and Kanad Ghose. hFS: A hybrid file system prototype for improving small file and metadata performance. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[ZRG14] Qing Zheng, Kai Ren, and Garth Gibson. Batchfs: Scaling the file system control plane with client-funded metadata servers. In *Proceedings of the 9th Parallel Data Storage Workshop (PDSW)*, 2014.