Organizing, Indexing, and Searching Large-Scale File Systems

Technical Report UCSC-SSRC-09-09
December 2009

Andrew W. Leung
aleung@cs.ucsc.edu

Storage Systems Research Center
Baskin School of Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064
http://www.ssrc.ucsc.edu/

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

ORGANIZING, INDEXING, AND SEARCHING LARGE-SCALE FILE SYSTEMS

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Andrew W. Leung

December 2009

The Dissertation of
Andrew W. Leung is approved:

Ethan L. Miller, Chair

Darrell D.E. Long

Garth R. Goodson

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Copyright © by
Andrew W. Leung
2009

Table of Contents

List of Figures vi

List of Tables xiii

Abstract xvi

Dedication xviii

Acknowledgments xix

1 Introduction 1
- 1.1 Main Contributions . . . 3
- 1.2 Analysis of Large-Scale File System Properties . . . 4
- 1.3 New Approaches to File Indexing . . . 5
- 1.4 Towards Searchable File Systems . . . 6
- 1.5 Organization . . . 7

2 Background and Related Work 8
- 2.1 The Growing Data Volume Trend . . . 8
- 2.2 Scalable Storage Systems . . . 9
  - 2.2.1 Parallel File Systems . . . 10
  - 2.2.2 Peer-to-Peer File Systems . . . 12
  - 2.2.3 Other File Systems . . . 13
- 2.3 The Data Management Problem in Large-Scale File Systems . . . 15
  - 2.3.1 Search-Based File Management . . . 18
  - 2.3.2 File System Search Background . . . 18
  - 2.3.3 Large-Scale File System Search Use Cases . . . 20
- 2.4 Large-Scale File System Search Challenges . . . 23
- 2.5 Existing Solutions . . . 26
  - 2.5.1 Brute Force Search . . . 26
  - 2.5.2 Desktop and Enterprise Search . . . 27
  - 2.5.3 Semantic File Systems . . . 28
  - 2.5.4 Searchable File Systems . . . 31

2.5.5 Ranking Search Results . . . 31
2.6 Index Design . . . 32
2.6.1 Relational Databases . . . 33
2.6.2 Inverted Indexes . . . 36
2.6.3 File System Optimized Indexing . . . 40
2.7 Summary . . . 41

3 Properties of Large-Scale File Systems 42
3.1 Previous Workload Studies . . . 45
3.1.1 The Need for a New Study . . . 46
3.1.2 The CIFS Protocol . . . 47
3.2 Workload Tracing Methodology . . . 47
3.3 Workload Analysis . . . 49
3.3.1 Terminology . . . 49
3.3.2 Workload Comparison . . . 50
3.3.3 Comparison to Past Studies . . . 52
3.3.4 File I/O Properties . . . 61
3.3.5 File Re-Opens . . . 63
3.3.6 Client Request Distribution . . . 65
3.3.7 File Sharing . . . 65
3.3.8 File Type and User Session Patterns . . . 69
3.4 Design Implications . . . 73
3.5 Previous Snapshot Studies . . . 74
3.6 Snapshot Tracing Methodology . . . 76
3.7 Snapshot Analysis . . . 77
3.7.1 Spatial Locality . . . 77
3.7.2 Frequency Distribution . . . 78
3.8 Summary . . . 81

4 New Approaches to File Indexing 82
4.1 Metadata Indexing . . . 84
4.1.1 Spyglass Design . . . 85
4.1.2 Hierarchical Partitioning . . . 86
4.1.3 Partition Versioning . . . 92
4.1.4 Collecting Metadata Changes . . . 95
4.1.5 Distributed Design . . . 96
4.2 Content Indexing . . . 97
4.2.1 Index Design . . . 97
4.2.2 Query Execution . . . 100
4.2.3 Index Updates . . . 101
4.2.4 Additional Functionality . . . 102
4.2.5 Distributing the Index . . . 103
4.3 Experimental Evaluation . . . 104

4.3.1 Implementation Details . . . 105
4.3.2 Experimental Setup . . . 106
4.3.3 Metadata Collection Performance . . . 107
4.3.4 Update Performance . . . 108
4.3.5 Space Overhead . . . 109
4.3.6 Selectivity Impact . . . 110
4.3.7 Search Performance . . . 111
4.3.8 Index Locality . . . 113
4.3.9 Versioning Overhead . . . 117
4.4 Summary . . . 118

5 Towards Searchable File Systems 119
5.1 Background . . . 121
5.1.1 Search Applications . . . 121
5.1.2 Integrating Search into the File System . . . 123
5.2 Integrating Metadata Search . . . 124
5.2.1 Metadata Clustering . . . 125
5.2.2 Indexing Metadata . . . 129
5.2.3 Query Execution . . . 132
5.2.4 Cluster-Based Journaling . . . 133
5.3 Integrating Semantic File System Functionality . . . 134
5.3.1 Copernicus Design . . . 135
5.3.2 Graph Construction . . . 137
5.3.3 Cluster Indexing . . . 139
5.3.4 Query Execution . . . 140
5.3.5 Managing Updates . . . 141
5.4 Experimental Results . . . 142
5.4.1 Implementation Details . . . 142
5.4.2 Microbenchmarks . . . 144
5.4.3 Macrobenchmarks . . . 150
5.5 Summary . . . 155

6 Future Directions 157
6.1 Large-Scale File Systems Properties . . . 157
6.2 New Approaches to File Indexing . . . 158
6.3 Towards Searchable File Systems . . . 159

7 Conclusions 161

Bibliography 165

List of Figures

2.1 **Network-attached storage architecture.** Multiple clients utilize a centralized server for access to shared storage. 9

2.2 **Parallel file system architecture.** Clients send metadata requests to metadata servers (MDSs) and data requests to separate data stores. 11

2.3 **Inverted index architecture.** The dictionary is a data structure that maps keywords to posting lists. Posting lists contain postings that describe each occurrence of the keyword in the file system. Searches require retrieving the posting lists for each keyword in the query. 37

3.1 **Request distribution over time.** The frequency of all requests, read requests, and write requests are plotted over time. Figure 3.1(a) shows how the request distribution changes for a single week in October 2007. Here request totals are grouped in one hour intervals. The peak one hour request total for corporate is 1.7 million and 2.1 million for engineering. Figure 3.1(b) shows the request distribution for a nine week period between September and November 2007. Here request totals are grouped into one day intervals. The peak one day intervals are 9.4 million for corporate and 19.1 million for engineering. 53

3.2 **Sequential run properties.** Sequential access patterns are analyzed for various sequential run lengths. The X-axes are given in a log-scale. Figure 3.2(a) shows the length of sequential runs, while Figure 3.2(b) shows how many bytes are transferred in sequential runs. 56

3.3 **Sequentiality of data transfer**. The frequency of sequentiality metrics is plotted against different data transfer sizes. Darker regions indicate a higher fraction of total transfers. Lighter regions indicate a lower fraction. Transfer types are broken into read-only, write-only, and read-write transfers. Sequentiality metrics are grouped by tenths for clarity. 57

3.4 **File size access patterns**. The distribution of open requests and bytes transferred are analyzed according to file size at open. The X-axes are shown on a log-scale. Figure 3.4(a) shows the size of files most frequently opened. Figure 3.4(b) shows the size of files from which most bytes are transferred. 59

3.5 **File open durations.** The duration of file opens is analyzed. The X-axis is presented on a log-scale. Most files are opened very briefly, although engineering files are opened slightly longer than corporate files. 60

3.6 **File lifetimes.** The distributions of lifetimes for all created and deleted files are shown. Time is shown on the X-axis on a log-scale. Files may be deleted through explicit delete request or truncation. 61

3.7 **File I/O properties**. The burstiness and size properties of I/O requests are shown. Figure 3.7(a) shows the I/O inter-arrival times. The X-axis is presented on a log-scale. Figure 3.7(b) shows the sizes of read and write I/O. The X-axis is divided into 8 KB increments. 62

3.8 **File open properties**. The frequency of and duration between file re-opens is shown. Figure 3.8(a) shows how often files are opened more than once. Figure 3.8(b) shows the time between re-opens and time intervals on the X-axis are given in a log-scale. 64

3.9 **Client activity distribution** The fraction of clients responsible for certain activities is plotted. 65

3.10 **File sharing properties**. We analyze the frequency and temporal properties of file sharing. Figure 3.10(a) shows the distribution of files opened by multiple clients. Figure 3.10(b) shows the duration between shared opens. The durations on the X-axis are in a log-scale. 66

3.11 **File sharing access patterns.** The fraction of read-only, write-only, and read-write accesses are shown for differing numbers of sharing clients. Gaps are seen where no files were shared with that number of clients. 67

3.12 **User file sharing equality**. The equality of sharing is shown for differing numbers of sharing clients. The Gini coefficient, which measures the level of equality, is near 0 when sharing clients have about the same number of opens to a file. It is near 1 when clients unevenly share opens to a file. 68

3.13 **File type popularity.** The histograms on the right show which file types are opened most frequently. Those on the left show the file types most frequently read or written and the percentage of accessed bytes read for those types. 70

3.14 **File type access patterns** The frequency of access patterns are plotted for various file types. Access patterns are categorized into 18 groups. Increasingly dark regions indicate higher fractions of accesses with that pattern. 71

3.15 **Examples of Locality Ratio.** Directories that recursively contain the ext attribute value `html` are black and gray. The black directories contain the value. The Locality Ratio of ext value `html` is 54% ($= 7/13$) in the first tree and 38% ($= 5/13$) in the second tree. The value of `html` has better spatial locality in the second tree than in the first one. 78

3.16 **Attribute value distribution examples.** A rank of 1 represents the attribute value with the highest file count. The linear curves on the log-log scales in Figures 3.16(a) and 3.16(b) indicate a Zipf-like distribution, while the flatter curve in Figure 3.16(c) indicates a more even distribution. 80

4.1 **Spyglass overview.** Spyglass resides within the storage system. The crawler extracts file metadata, which gets stored in the index. The index consists of a number of partitions and versions, all of which are managed by a caching system. 86

4.2 **Hierarchical partitioning example.** Sub-tree partitions, shown in different colors, index different storage system sub-trees. Each partition is stored sequentially on disk. The Spyglass index is a tree of sub-tree partitions. 87

4.3 **Signature file example.** Signature files for the ext and size metadata attributes are shown. Each bit corresponds to a value or set of values in the attribute value space. One bits indicate that the partition *may* contain files with that attribute value while zero bits that they definitely do not. In the top figure, each bit corresponds to an extension. False-positives occur in cases where multiple extensions hash to the same bit position. In the bottom figure, each bit corresponds to a range of file sizes. 90

4.4 **Versioning partitioning example.** Each sub-tree partition manages its own versions. A baseline index is a normal partition index from some initial time T_0. Each incremental index contains the changes required to roll query result forward to a new point in time. Each sub-tree partition manages its version in a version vector. 93

4.5 **Snapshot-based metadata collection.** In snapshot 2, block 7 has changed since snapshot 1. This change is propagated up the tree. Because block 2 has not changed, we do not need to examine it or any blocks below it. 95

4.6 **Indirect Index Design.** The indirect index stores the dictionary for the entire file system and each keyword's posting lists contain locations of partition segments. Each partition segment is kept sequential on-disk. 98

4.7 **Segment Partitioning.** The namespace is broken into partitions that represent disjoint sub-trees. Each partition maintains posting list segments for keywords that occur within its sub-trees. Since each partition is relatively small, these segments can be kept sequential on-disk. 99

4.8 **Metadata collection performance.** We compare Spyglass's snapshot-based crawler (SB) to a straw-man design (SM). Our crawler has good scalability; performance is a function of the number of changed files rather than system size. 107

4.9 **Update performance.** The time required to build an initial baseline index shown on a log-scale. Spyglass updates quickly and scales linearly because updates are written to disk mostly sequentially. 108

4.10 **Space overhead.** The index disk space requirements shown on a log-scale. Spyglass requires just 0.1% of the Web and Home traces and 10% of the Eng trace to store the index. 109

4.11 **Comparison of Selectivity Impact.** The selectivity of queries in our query set is plotted against the execution time for that query. We find that query performance in Spyglass is much less correlated to the selectivity of the query predicates than the DBMSs, which are closely correlated with selectivity. . . . 110

4.12 **Query set run times.** The total time required to run each set of queries. Each set is labeled 1 through 3 and is clustered by trace file. Each trace is shown on a separate log-scale axis. Spyglass improves performance by reducing the search space to a small number of partitions, especially for query sets 2 and 3, which are localized to only a part of the namespace. 112

4.13 **Query execution times.** A CDF of query set execution times for the Eng trace. In Figures 4.13(b) and 4.13(c), all queries are extremely fast because these sets include a path predicate that allows Spyglass to narrow the search to a few partitions. 114

4.14 **Index locality.** A CDF of the number of partitions accessed and the number of accesses that were cache hits for our query set. Searching multiple attributes reduces the number of partition accesses and increases cache hits. 115

4.15 **Impact of signature file size.** The percent of one bits in signature files for four different attributes is shown. The percent of one bits decreases as signature file size increases. However, even in the case where signature files are only 1 KB less than 4% of total bits are set to one. As mentioned earlier, the size and atime attributes use special hashing functions that treat each bit as a range rather than a discrete value. This approach keeps the fraction of one bits below 1% in all cases. 116

4.16 **Versioning overhead.** The figure on the left shows total run time for a set of 450 queries. Each version adds about 10% overhead. On the right, a CDF shows per-query overheads. Over 50% of queries have an overhead of 5 ms or less. . . 117

5.1 **File search applications.** The search application resides on top of the file system and stores file metadata and content in separate search-optimized indexes. Maintaining several large index structures can add significant space and time overheads. 122

5.2 **Metadata clustering.** Each block corresponds to an inode on disk. Shaded blocks labeled 'D' are directory inodes while non-shaded blocks labeled 'F' are file inodes. In the top disk layout, the indirection between directory and file inodes causes them to be scattered across the disk. The bottom disk layout shows how metadata clustering co-locates inodes for an entire sub-tree on disk to improve search performance. Inodes reference their parent directory in Magellan; thus, the pointers are reversed. 126

5.3 **Inode indexing with a K-D tree.** A K-D tree is shown with nodes that are directory inodes are shaded with a 'D'. File inodes are not shaded and labeled 'F'. K-D trees are organized based on attribute value not namespace hierarchy. Thus, a file inode can point to other file inodes, *etc*. The namespace hierarchy is maintained by inodes containing the inode number of their parent directory. Extended attributes, such a title and file type are included in the inode. 130

5.4 **Copernicus overview.** Clusters, shown in different colors, group semantically related files. Files within a cluster form a smaller graph based on how the files are related. These links, and the links between clusters, create the Copernicus namespace. Each cluster is relatively small and is stored in a sequential region on disk for fast access. 137

5.5 **Cluster indexing performance.** Figure 5.5(a) shows the latencies for balanced and unbalanced K-D tree queries, brute force traversal, and inserts as cluster size increases. A balanced K-D tree is the fastest to search and inserts are fast even in larger clusters. Figure 5.5(b) shows latencies for K-D tree rebalancing and disk writes. Rebalancing is slower because it is requires $O(N \times \log N)$ time. 145

5.6 **Create performance.** The throughput (creates/second) is shown for various system sizes. Magellan's update mechanism keeps create throughput high because disk writes are mostly to the end of the journal, which yields good disk utilization. Throughput drops slightly at larger sizes because more time is spent searching clusters. 147

5.7 **Metadata clustering.** Figure 5.7(a) shows create throughput as maximum cluster size increases. Performance decreases with cluster size because inode caching and Bloom filters become less effective and K-D tree operations become slower. Figure 5.7(b) shows that query performance is worse for small and large sizes. 149

5.8 **Metadata workload performance comparison.** Magellan is compared to the Ceph metadata server using four different metadata workloads. In all cases, both provide comparable performance. Performance differences are often due to K-D tree utilization. 151

5.9 **Query execution times.** A CDF of query latencies for our three query sets. In most cases, query latency is less than a second even as system size increases. Query set 2 performs better than query set 1 because it includes a directory path from where Magellan begins the search, which rules out files not in that sub-tree from the search space. 153

5.10 **Fraction of clusters queried.** A CDF of the fraction of clusters queried for query set 1 on our three data sets is shown. Magellan is able to leverage Bloom filters to exploit namespace locality and eliminate many clusters from the search space. 154

List of Tables

3.1 **Summary of observations.** A summary of important file system trace observations from our trace analysis. Note that we define clients to be unique IP addresses, as described in Section 3.3.1. 44

3.2 **Summary of major file system workload studies over the past two decades.** For each study, we show the date of trace collection, the file system or protocol studied, whether it involved network file systems, and the workloads studied. . 45

3.3 **Summary of trace statistics.** File system operations broken down by workload. All operations map to a single CIFS command except for file stat (composed of `query_path_info` and `query_file_info`) and directory read (composed of `find_first2` and `find_next2`). Pipe transactions map to remote IPC operations. 51

3.4 **Comparison of file access patterns.** File access patterns for the corporate and engineering workloads are compared with those of previous studies. CAMPUS and EECS [48] are university NFS mail server and home directory workloads, respectively. Both were measured in 2001. Sprite [17], Ins and Res [137] are university computer lab workloads. Sprite was measured in 1991 and Ins and Res were measured between 1997 and 2000. NT [177] is a combination of development and scientific workloads measured in 1998. 54

3.5 **Summary of major file system snapshot studies over the past two decades.** For each study, the date of trace collection, the file system or protocol studied, whether it involved network file systems, and the kinds of workloads it hosted are shown. 75

3.6 **Metadata traces collected.** The small server capacity of the Eng trace is due to the majority of the files being small source code files: 99% of files are less than 1 KB. 76

3.7 **Attributes used.** We analyzed the fields in the inode structure and extracted ext values from path. 76

3.8 **Locality Ratios of the 32 most frequently occurring attribute values.** All Locality Ratios are well below 1%, which means that files with these attribute values are recursively contained in less than 1% of directories. 79

4.1 **Use case examples.** Metadata search use cases collected from our user survey. The high-level questions being addressed are on the left. On the right are the metadata attributes that are being searched and example values. Users used basic inode metadata as well as specialized extended attributes, such as legal retention times. Common search characteristics include multiple attributes, localization to part of the namespace, and "back-in-time" search. 85

4.2 **Query Sets.** A summary of the searches used to generate our evaluation query sets. 111

4.3 **Query throughput.** We use the results from Figure 4.12 to calculate query throughput (queries per second). We find that Spyglass can achieve query throughput that enables fast metadata search even on large-scale storage systems. 113

5.1 **Inode attributes used.** The attributes that inodes contained in our experiments. 143

5.2 **Metadata workload details.** 150

5.3 **Metadata throughput (ops/sec).** The Magellan and Ceph throughput for the four workloads. 151

5.4 **Query Sets.** A summary of the searches used to generate our evaluation query sets. 152

Abstract

Organizing, Indexing, and Searching Large-Scale File Systems

by

Andrew W. Leung

The world is moving towards a digital infrastructure. This move is driving the demand for data storage and has already resulted in file systems that contain petabytes of data and billions of files. In the near future file systems will be storing exabytes of data and trillions of files. This data growth has introduced the key question of how we effectively find and manage data in this growing sea of information. Unfortunately, file organization and retrieval methods have not kept pace with data volumes. Large-scale file systems continue to rely on hierarchical namespaces that make finding and managing files difficult.

As a result, there has been an increasing demand for search-based file access. A number of commercial file search solutions have become popular on desktop and small-scale enterprise systems. However, providing effective search and indexing at the scale of billions of files is not a simple task. Current solutions rely on general-purpose index designs, such as relational databases, to provide search. General-purpose indexes can be ill-suited for file system search and can limit performance and scalability. Additionally, current search solutions are designed as applications that are separate from the file system. Providing search through a separate application requires file attributes and modifications to be replicated into separate index structures, which presents consistency and efficiency problems at large-scales.

This thesis addresses these problems through novel approaches to organizing, indexing, and searching files in large-scale file systems. We conduct an analysis of large-scale file system properties using workload and snapshot traces to better understand the kinds of data being stored and how it is used. This analysis represents the first major workload study since 2001 and the first major study of enterprise file system contents and workloads in over a decade. Our analysis shows a number of important workload properties have changed since previous studies (*e. g.*, read to write byte ratios have decreased to 2:1 from 4:1 or higher in past studies) and examines properties that are relevant to file organization and search. Other important observations

include highly skewed workload distributions and clustering of metadata attribute values in the namespace.

We hypothesize that file search performance and scalability can be improved with file system specific index solutions. We present the design of new file metadata and file content indexing approaches that exploit key file system properties from our study. These designs introduce novel file system optimized index partitioning, query execution, and versioning techniques. We show that search performance can be improved up to 1–4 orders of magnitude compared to traditional approaches. Additionally, we hypothesize that directly integrating search into the file system can address the consistency and efficiency problems with separate search applications. We present new metadata and semantic file system designs that introduce novel disk layout, indexing, and updating methods to enable effective search without degrading normal file system performance. We then discuss on going challenges and how this work may be extended in the future.

To my family for all of their love and support.

Acknowledgments

My career in graduate school would not have been possible without the help, guidance, and friendship of many. I first want to thank my adviser Ethan L. Miller who is the main reason I decided to pursue a Ph.D. He taught me the important research questions to ask and how to critically analyze a problem. He was always willing to help me, whether it was with research, giving talks and presentations, or looking for a job. Most importantly he taught me how to make research an enjoyable and fulfilling experience. I want to thank him for all of the late night hours he spent helping me with paper submissions when I am sure that sleep was far more appealing. Also, his knack for general trivia proved extremely useful during multiple trivia nights at 99 Bottles.

I would also like to thank my other thesis committee members, Darrell Long and Garth Goodson. Their willingness to listen to my wild ideas and help me turn them into actual meaningful research was critical in helping me complete this thesis. I also want to thank Erez Zadok for his very helpful and detailed comments on my advancement proposal.

I want to thank my friends and colleagues in the Storage Systems Research Center (SSRC) and NetApp's Advanced Technology Group (ATG) who made my graduate career, despite the long hours and hard work, so enjoyable. My internship experience at NetApp's ATG was one the most valuable parts of my graduate school success. I want to thank my mentors and colleagues in the ATG, including Shankar Pasupathy, Garth Goodson, Minglong Shao, Timothy Bisson, Kiran Srinivasan, Kaladhar Voruganti, and Stephen Harpster. They provided wonderful guidance and I was surprised by how eager they were to see me succeeded. Additionally, they provided me with access to file system trace data that is a major part of this thesis.

My friends and lab mates in the SSRC were one of the main reasons I decided to stay for a Ph.D. They made staying in the lab fun and provided encouragement during the long hours before paper or presentation deadlines. Though they are numerous, I would like to thank Eric Lalonde, Mark Storer, Kevin Greenan, Jonathan Koren, Jake Telleen, Stephanie Jones, Ian Adams, Christina Strong, Aleatha Parker-Wood, Keren Jin, Deepavali Bhagwat, Alex Nelson, Damian Eads, and Sage Weil for putting up with me and helping me with my research. The

numerous barbecues, video game breaks, and late night drinks at a bar or at someone's house were a much needed and appreciated distraction from work.

During graduate school I was lucky to avoid the headaches that other student have worrying about their funding sources. I was fortunate, especially during these current economic times, to have a number of funding sources for my research. I want to thank my these funding sources, which include the Department of Energy's Petascale Data Storage Institute under award DE-FC02-06ER25768, the National Science Foundation under award CCF-0621463, and the SSRC's industrial sponsors.

Finally, I would like to thank my family; my mom, my dad, and my sister Allison. Without them none of this would have been possible. They provided me with financial help, a place to come home to on weekends with free food and laundry, an outlet for me to vent about work and all with unwavering love and support. Their encouragement helped me through multiple stressful and difficult times. My sister's tireless copy editing caught many of my typos, incorrect grammar, and generally poor writing. I dedicate this thesis to them.

Chapter 1

Introduction

Today there is a need for data storage like never before and this need is only increasing. Modern businesses, governments, and people's daily lives have shifted to a digital infrastructure, which is projected to require close to two zettabytes (two million petabytes) of storage by 2011 [58]. This demand, coupled with improvements in hard disk capacities and network bandwidth, has yielded file systems that store petabytes of data, billions of files, and serve data to thousands of users [46, 60, 91, 139, 146, 180, 183]. File systems of this scale introduce a new challenge; How do we organize so much data so that it is easy to find and manage? These systems store data that make up the backbone of today's digital world and it is paramount that data can be effectively organized and retrieved. Additionally this problem is pervasive, impacting scientific [70, 72], enterprise [51], cloud [126], and personal [142] storage.

File systems of this scale make effectively finding and managing data extremely difficult. Current file organization methods are based on designs that are over forty years old and designed for systems with less than 10 MB of data [38]. These systems use a basic hierarchical namespace that is restrictive, difficult to use, and limits scalability. These systems require users to manage and navigate huge hierarchies with billions of files, which wastes significant time and can lead to misplaced or permanently lost data. Similarly, users and administrators must rely on slow, manual tools to try and understand the data they manage, which can leads to under utilized or poorly managed systems. The basic problem is that *file systems have grown beyond*

the scope of basic file organization and retrieval methods, which has decreased our ability to effectively find, manage, and utilize data.

This growing data management problem has lead to an increased emphasis on search-based file access. File search involves indexing attributes derived from file metadata and content, allowing them to be used for retrieval. File search is useful in large-scale file systems because it allows users to specify *what* they want rather than *where* it is. For example, a user can access his document by just knowing it was accessed in the last week and deals with the quarterly financial records. Additionally, finding files to migrate to secondary storage requires only knowing the kinds of files to be migrated (*e. g.*, files larger than 50 MB that have not been accessed in 6 months), instead of where all of these files are located. In most cases, a file's location is irrelevant; users need to retrieve their data using whatever information they may have about it. Moreover, search allows complex, ad hoc questions to be asked about the files being stored that help to locate, manage, and analyze data. The effectiveness of search-based file access has led to many file system search applications to become commercially available for desktop [14, 66, 110] and small-scale enterprise [55, 67, 85] file systems. File system search has also become a popular research area [49, 63, 69, 124, 154].

Unfortunately, enabling fast and effective search in large-scale file systems is very difficult. Current solutions, which are designed as applications outside of the file system and which rely on general purpose index designs, are too expensive, slow, and cumbersome to be effective in large-scale systems. The index is the data structure that enables effective search functionality. General-purpose indexes, such as relational databases, are not designed for file system search and can be ill-suited to address file system search needs. As Stonebraker *et al.* state, there is "a substantial performance advantage to specialized architectures" [161] because general-purpose solutions make few specialized optimizations and can have mismatched or unused functionality, which limit their performance and scalability.

Additionally, while increasingly important, file search is provided by an application that is outside of the file system, not the file system itself. Separating the two is an odd model given that search applications and file systems share a common goal: organization and retrieval of files. Implementing search functionality in an application outside of the file system leads

to duplication functionality and requires metadata attributes and changes to be replicated in a separate database or search engine. Maintaining two indexes of file attributes (*e. g.*, the file system's and the search application's index), leads to many consistency and efficiency issues, particularly at the scale of billions of files. As a result, users and administrators continue to struggle to find, organize, and manage data in large-scale file systems.

1.1 Main Contributions

In this thesis we address this problem by improving how files are organized, indexed, and searched in large-scale file systems. This thesis examines two key hypotheses. First, search performance and scalability can be improved with new indexing structures that leverage file system specific properties. Second, it is possible to enable efficient search directly in the file system without degrading normal file system performance. These two hypotheses are evaluated in the three key contributions of this thesis:

(1) We measure and analyze workloads and metadata snapshots from several large-scale file systems used in real world deployments. We conduct a number of studies to compare how workloads have changed since previous studies, as well as, perform several new experiments. Our analysis reveals a number of observations that are relevant to better file organization and indexing and to file system design in general.

(2) Using observations from our file system analysis, we present the design of two new indexing structures for file metadata and content. Our designs exploit file attribute locality and distribution properties to improve performance and scalability. Additionally, new approaches to index updating and versioning are used. An evaluation shows that search performance can be improved between 1–4 orders of magnitude compared to traditional solutions.

(3) We then present two novel file system designs that directly integrate search functionality, eliminating the need for external search applications. The first organizes metadata so that it can be easily and effectively queried while maintaining good performance for normal metadata workloads. The second is a new approach to semantic file system design that organizes the

namespace into graph-based structure. This new index structure allows dynamic, search-based file access and navigation using inter-file relationships.

The following sections detail the individual thesis contributions.

1.2 Analysis of Large-Scale File System Properties

Designing better methods for organizing and managing files requires first understanding the properties of the files being stored and how they are used. This understanding is often guided by measuring and analyzing file system workload and snapshot traces. Trace-based file system studies have guided the designs of many past file systems [88, 116, 138]. For example, caching in the Sprite network file system [116] was guided by the observation that even small client-side caches can be effective for improving performance [123]. Similarly, the log-structured file system [138] was guided by observations that network file system workloads are becoming increasingly write-oriented [17, 123] due to the presence of client-side caching.

We collect and analyze traces of file system workloads and contents from several large-scale network file servers deployed in the NetApp corporate data center. These servers were used by thousands of employees from multiple departments. Our analysis focuses on trends since past studies, conducts a number of new experiments, and looks at the impact on file organization and indexing. Our study represents the first major workload study since 2001 [48], the first to look at large-scale CIFS [92] network workloads, and the first major study of enterprise file server contents and workloads in over ten years.

Our analysis reveals a number of interesting observations, such as workloads are becoming increasingly write-heavy, files are increasing in size, and have longer lifetimes compared to previous studies. Additionally, we find that file access is mostly transient. Only 66% of opened files are re-opened and 95% are re-opened less than five times. Files are also rarely shared as 76% of files are never opened by more than one client. Workload distribution is heavily skewed with only 1% of clients accouting for almost 50% of file requests. Similarly, metadata attribute distributions are highly skewed and follow the power-law distribution [152].

Also, metadata attribute values are heavily clustered in the namespace. All metadata value that we studied occurred in fewer than 1% of total directories.

1.3 New Approaches to File Indexing

Fast and effective search in large-scale file systems is difficult to achieve. The index is the data structure that enables search functionality and ineffective index design can limit performance and scalability. Current file search solutions utilize general-purpose indexing methods that are not designed for file systems. These indexes were designed for other workloads, have few file system search optimizations, and have extra functionality that is not needed for file system search. For example, file metadata is often indexed using relational database management systems (DBMSs). However, DBMSs are designed for on-line transaction processing workloads [16], use locking and transactions that can add overhead [165], and are not a perfect fit for metadata search [162].

We hypothesize that new index designs that leverage file system properties can improve performance and scalability. We propose two new index designs, one for structured metadata search, called Spyglass, and one for unstructured content search, that leverage observations from our trace-based analysis to improve performance. For example, we found that metadata attribute values exhibit *spatial locality*, which means that they tend to be highly clustered in the namespace. Thus, files owned by user `Andrew` are often clustered in locations such as the user's home directory or active project directories and are not scattered across the namespace. We introduce the notion of *hierarchical partitioning*, which allows the index to exploit spatial locality by partitioning and allowing fine-grained index control based on the namespace. Hierarchical partitioning makes it possible for searches, updates, and caching to be localized to only the relevant parts of the namespace. We also introduce a number of other techniques that improve query execution, update and versioning operation, and metadata collection. An evaluation of our metadata index prototype shows that search performance can be improved up to 1–4 orders of magnitude compared to basic DBMS setups, while providing update performance that is up to $40\times$ faster and requiring less than 0.1% of total disk space.

1.4 Towards Searchable File Systems

Despite the increasing trend towards search becoming a primary way to access and manage files, file systems do not provide any search functionality. Current file system hierarchies can only be searched with brute force methods, such as `grep` and `find`. Instead, a separate search application that maintains search-based index structures and is separate from the file system is often used [14, 55, 67]. However, search applications and the file system share the same goal: organizing and retrieving files. Keeping file search separate from the file system leads to consistency and efficiency issues as all file attributes and changes must be replicated in separate applications, which can limit performance and usability especially at large-scales.

We hypothesize that a more complete, long-term solution is to integrate search functionality directly into the file system. Doing so eliminates the need to maintain a secondary search application, allows file changes to be searched in real-time, and allows data organization to correspond to the need for search functionality. However, enabling effective search within the file system has a number of challenges. First, there must be an way to organize file attributes internally so that they can be efficiently searched and updated. Second, this organization must not significantly degrade performance for normal file system workloads.

We propose two new file system designs that directly integrate search. The first, Magellan, is a new metadata architecture for large-scale file systems that organizes the file system's metadata so that it can be efficiently searched. Unlike previous work, Magellan does not use relational databases to enable search. Instead, it uses new query-optimized metadata layout, indexing, and journaling techniques to provide search functionality and high performance in a single metadata system. In Magellan, all metadata look ups, including directory look ups, are handled using a single search structure, eliminating the redundant index structures that plague existing file systems with search grafted on.

The second, Copernicus, is a new semantic file system design that provides a search-based namespace. Unlike previous semantic file systems which were designed as naming layers above a traditional file system or general-purpose index, Copernicus uses a dynamic, graph-based index that stores file attributes and relationships. This graph replaces the traditional

directory hierarchy and allows the construction of dynamic namespaces. The namespace allows "virtual" directories that correspond to a query and navigation to be done using inter-file relationships. An evaluation of our Magellan prototype shows that it is capable of searching millions of files in under a second, while providing metadata performance that is comparable to, and sometimes better than, other large-scale file systems.

1.5 Organization

This thesis is organized as follows:

Background and related work: Chapter 2 outlines the file retrieval and management problems caused by very large data volumes. We also provide the necessary background information on basic file system search concepts and discuss why large-scale file systems make search difficult. Then, we discuss why existing solutions do not address these difficulties.

Properties of large-scale file systems: Chapter 3 presents the measurement and analysis of large-scale file system workloads and snapshots. We compare our findings with previous studies, conduct several new experiments, and discuss how our findings impact how file systems organize and manage files.

New approaches to file indexing: Chapter 4 presents new index designs that exploit the file system properties that we observed in Chapter 3. We discuss index designs for file metadata and content and compare performance to general-purpose DBMS solutions.

Towards searchable file systems: Chapter 5 discusses how search can be integrated directly into the file system. We present the design of a hierarchical file system metadata architecture and a semantic file system that use search-optimized organization and indexing.

Future directions: Chapter 6 discusses the future directions of this work. We present ways that current work can be extended and the new research directions that this work enables.

Conclusions: Chapter 7 summarizes our findings and concludes this thesis.

Chapter 2

Background and Related Work

This chapter discusses the rapid and continuing growth of data volumes and the impact it has on file and data management. To address this challenge we motivate the use of search-based file access in large-scale file systems. We then provide a basic introduction to file system search concepts and discuss the challenges in enabling such access in large-scale file systems.

2.1 The Growing Data Volume Trend

The digital universe is rapidly expanding [58]. Many aspects of business, science, and daily life are becoming increasingly digital, producing a wealth of data that must be stored. For example, today's media such as photos and videos are mostly digital. Web and cloud services that host this data must be able to store and serve an indefinite amount of this data. Facebook must manage over 60 billion image files and store over 25 TB of new photo data every week [52]. It is expected that CERN's Large Hadron Collider will annually produce over 15 PB of data [42]. In another example, government mandates such as HIPAA [175] and Sarbanes-Oxley [176] require the digitization and retention of billions of medical and financial records. In 2007 the digital universe (*i. e.*, the total number of bytes created, captured, and replicated) was estimated to be 281 exabytes and is expected to grow ten fold by 2011 to about 2.8 zettabytes [58].

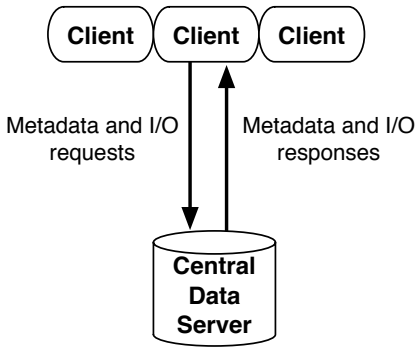


Figure 2.1: **Network-attached storage architecture.** Multiple clients utilize a centralized server for access to shared storage.

The value of and requirements being place on the data being generated are also increasing. Many important aspects of society are now dependent on the ability to effectively store, locate, and utilize this data. Some data, such as oil and gas survey results, can be worth millions of dollars [179]. Other data, such as government department of defense files or nuclear test results can be vital to national security and data such as genome sequences and biomolecular interactions are key to the future of modern medicine. Additionally, the digitization of people's personal lives (*e. g.*, personal photos, letters, and communications) has given data great sentimental value and made its storage critical to preserving personal histories.

2.2 Scalable Storage Systems

The increasing demand for data storage has driven the design of data storage systems for decades. The scale at which data is produced has forced system designs to focus on scalability and eliminating bottlenecks that may limit performance. There have been major improvements in throughput and latency, reliability, cost-effectiveness, and distributed designs over the years. As a result, today's large-scale file systems are capable of storing petabytes of data and billions of files, are composed of thousands of devices, and can serve data to thousands of users and applications [2, 36, 46, 60, 62, 91, 139, 146, 180, 183].

Early scalable file systems used a basic network-attached storage model where many clients were directly connected to a single centralized storage server as shown in Figure 2.1. Storing data on a centralized server allowed it to be more easily shared, accessed from multiple locations, and provided more efficient utilization of storage resources. Protocols such as AFS [79] and NFS [127] are used to transfer data between the client and server. A number of file systems have been designed for the centralized file server, such as LFS [138] and WAFL [77]. These systems are often optimized for write performance because client caches are able to reduce the read traffic seen by the server. However, a single centralized server can often become a performance bottleneck that limits scalability. A single server can only serve a limited amount of storage and can become overwhelmed with requests as the number of clients or requests increases. To address this problem, clustered file systems, such as Frangipani [172] and ONTAP GX [46], allow multiple central servers to be used.

2.2.1 Parallel File Systems

Parallel file systems are a type of clustered file system that can achieve better scalability by allowing storage devices to be highly cluster (*e. g.*, up to thousands of devices) and by separating the data and metadata paths. The basic design is illustrated in Figure 2.2. These systems are composed of metadata servers (MDSs) that store and handle file metadata requests and data servers that store and handle file data requests. Clients communicate directly to both metadata and data servers for file operations. Files are often striped across many storage devices allowing clients to access a file using many parallel I/O streams.

Network-Attached Secure Disk (NASD) [62] was an early parallel file system architecture that introduced the concept of object-based storage. Object-based storage exploits growing price reductions and performance improvements in commodity CPU and memory hardware to provide intelligent network-attached storage devices. These object-based storage devices (OSDs) greatly improve the design because MDSs need to do less management, as OSDs can manage their own local storage.

The original NASD design has spawned a number of other parallel file systems. The General Parallel File System (GPFS) [146] from IBM uses a single metadata server, called

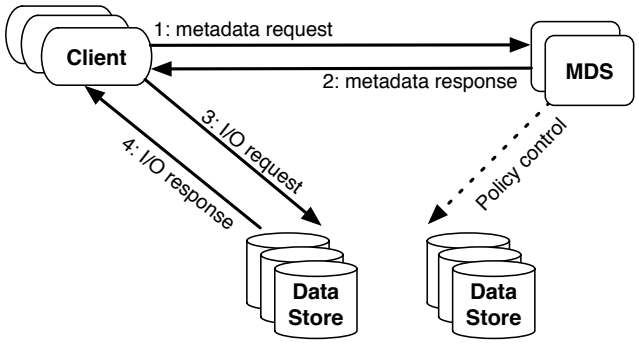


Figure 2.2: **Parallel file system architecture.** Clients send metadata requests to metadata servers (MDSs) and data requests to separate data stores.

the metanode, to handle metadata requests and perform distributed metadata locking. The distributed locking protocol hands out lock tokens to lock manager nodes, which eliminates the need for the metanode to handle all locking requests, thereby improving scalability. A fail-over mechanism allows another node to step in to perform metadata activities if the metanode fails.

Similarly, the Google File System (GFS) [60] uses a single metadata server, called the master, and many data servers, called chunkservers. By using a single master, the overall metadata design is simplified, however, they acknowledge it can present a performance bottleneck. To reduce load on the master, the master only stores two kinds of in-memory metadata, a chunk location mapping, and an operations log for crash recovery. This approach reduces the number of client requests made to the master. Also to improve performance, GFS uses a loose consistency model that is acceptable for their workloads. The Linux PVFS file system [34] also uses a single metadata server.

However, when metadata requests cannot be avoided, such as during a metadata intensive workload, a single metadata server can present a bottleneck. To alleviate this problem, PanFS [183] from Panasas uses a cluster of metadata servers, called manager nodes. Manager nodes, unlike other MDSs, do not store metadata in them. Instead, file metadata (*i. e.*, owner, size, modification time, *etc.*) is stored directly with the file's objects on the storage devices. The metadata manager manages the semantics for these files, such as locking and synchronization.

Additionally, it maintains an operations log for crash recovery. Each manager is responsible for managing a single logical volume, which is usually made up of about ten data nodes.

The Ceph file system [180] also clusters MDSs for performance, though uses a different approach. Rather than have the storage devices handle metadata requests for the data they store, the MDS cluster handles all metadata requests and uses the data stores only for persistent storage. The MDS cluster provides good load-balancing by allowing fragments of the namespace, as small as portions of a directory, to be replicated across the cluster using a method called dynamic sub-tree partitioning [182]. MDS load is decreased through a file mapping function [181] that maps inode identifiers to their location on storage devices, eliminating the need for the MDS to perform block look ups. Inodes are embedded directly within their parent directory, allowing more efficient reading and writing of metadata.

2.2.2 Peer-to-Peer File Systems

Another trend in large-scale file systems has been to look at how to more effectively scale costs by relying on commodity desktop computers or those available over the Internet instead of high-end, customized servers. These systems are often composed of less powerful machines, such as personal computers, that leverage cheap or freely available resources. Each node's role is dynamic and there is often no clear client-server model, causing these systems to often be referred to as peer-to-peer file systems. As the performance of commodity hardware increases to the point where it can rival custom hardware, peer-to-peer file systems are becoming increasingly common. These systems are popular for content distribution networks, such as BitTorrent and are beginning to move into enterprise environments [8, 41].

OceanStore [91] is a global-scale file system that provides data redundancy and security. Files are addressed using globally unique identifiers (GUIDs) and different storage mechanisms are used for active and archival data. A distributed routing algorithm locates files, using their GUID, that may be on servers scattered across the globe. Data can be placed on any node in the file system, which allows for flexible replication, caching, and migration. Since devices are far less reliable then dedicated machines, complex update algorithms must be used to ensure

basic file system semantics are preserved even when nodes frequently fail or may be prone to abuse.

Like OceanStore, PAST [139] is another global-scale file system where commodity nodes organize into a overlay network. PAST leverages the global distribution of nodes to improve the reliability and availability of data, as no single physical disaster will likely destroy all copies of a data object, making it an attractive solution for backup and archival storage. PAST uses the Pastry [140] overlay routing system, which takes into account geographical location when routing data.

Pangaea [141] is a peer-to-peer file system that is intended for day-to-day use rather then backup or archival storage. Pangaea uses the under-utilized resources of most computers on a LAN to provide high-performance file access that should resemble a local file system. This is done through pervasive replication, which makes copies of data whenever it is accessed, ensuring that data is always close to the users that are accessing it. To provide data consistency, replicas use an optimistic coordination protocol that allows them to efficiently exchange update messages.

FARSITE [2] attempts to address a similar problem as Pangaea, providing performance comparable to a local file system. FARSITE notes that, since desktop computers are being utilized, the system is likely to experience a much wider variety of errors and faults than traditional file systems. As a result, they use a byzantine fault-tolerant protocol to ensure that the system remains available even in the face of unpredictable errors. A distributed directory protocol [44] that allows the namespace to be distributed over many nodes using immutable tree-structure file identifiers. This approach eliminates the need to migrate files when their pathnames change because distribution is done using immutable file identifiers rather than pathname. Various kinds of metadata leases allow FARSITE to effectively balance metadata load to avoid bottlenecks.

2.2.3 Other File Systems

While basic scalability and performance are the focal points for many applications, a number of other file systems have been designed to enable other kinds of storage functionality.

BigTable [36] and Boxwood [102] provide logical abstractions above the physical storage that change the way applications interact with data. These systems are often intended to provide a better way to represent complex stored data rather than with a basic hierarchical namespace. BigTable is a large-scale multi-dimensional map where data is addressed using row, column, and time stamp identifiers rather than traditional file IDs. It is built on top of the Google file system and uses the Google SSTable, which is a file format that maps ordered key to values, to map keys to their data locations in the file system. Each row in the table is called a tablet and is the unit at which data is distributed across the file system. BigTable is intended to be used in large-scale data processing applications, such as building a web index. Similarly, Amazon's Dynamo [41] uses a basic key-value storage interface built on top of a peer-to-peer file system. Their focus is on providing effective Service Level Agreements (SLAs) to customers and applications.

Boxwood [102] allows the construction of general data structures over a cluster of storage servers. The storage system can be used as any other data structure would be used by an application. For instance, building a distributed database is significantly easier when the storage system presents distributed B-tree or hash table abstractions. Boxwood handles many of the underlying communication mechanisms that make developing traditional distributed systems difficult.

The need to preserve data for decades or centuries has resulted in the design of several large-scale archival file systems. It is predicted that archival storage demands will exceed primary storage demands as more data is created and kept [58]. Venti [130] is a content-addressable distributed archival storage system that seeks to ensure data is maintained through write-once storage. The namespace is the content hashes of the data blocks and is distributed across nodes in the system and uses disk-based storage rather than tape, which provides better reliability and performance. Similarly, Pergamum [166] uses disk storage, rather than tape and through the use of spin-down and data encoding techniques can provide cheaper, more reliable long-term storage.

2.3 The Data Management Problem in Large-Scale File Systems

The increasing amount of data being stored in enterprise, cloud and high performance file systems is changing the way we access and manage files. The improvements in file system design that we discussed have enabled the storage of petabytes of data and billions of files. However, this growth has resulted in a new and largely unsolved problem; How to effectively find and manage data at such large-scales. File systems organize files into a basic hierarchical namespace that was designed over forty years ago when file systems contained less than 10 MB of data [38]. File access requires explicit knowledge of the file's name and location. While the hierarchical namespace has been successful as file systems have grown to millions of files (as evidenced by its longevity), its limitations become obvious and very problematic as file systems reach billions of files. The basic problem is that *as file systems have grown in scale, improvements in file organization and retrieval have not kept pace resulting in no way to effectively find and manage files in large-scale file systems*.

The goal of the file system is to provide reliable, persistent storage of files and to organize them in a way that they can be easily accessed and retrieved. Stored data is of limited value if it cannot be effectively accessed and utilized. When file systems lack effective organization and retrieval functionality, data essentially becomes "trapped" within the file system, cannot be utilized, and is of little value. In some cases poor organization can cause files to be completely lost. For example, NASA's original footage of the moon landing has been permanently misplaced despite being stored on tape shortly after it was recorded [171].

Moreover, storage users and administrator waste considerable time trying to organize and locate data. At the petabyte-scale and beyond, basic tasks such as recalling specific file locations, finding which files consume the most disk space, or sharing files between users become difficult and time consuming. Additionally, there are a growing number of complicated tasks such as regulatory compliance, managing hosted files in a cloud, and migrating files between storage tiers for maximum efficiency that businesses must solve. For users, this wasted time can decrease productivity and limits the overall effectiveness of the file system. If a file's location is not know, a great deal of effort and time must be spent navigating and traversing the

namespace trying to locate it. This process becomes much more difficult, time consuming, and less accurate when users must sift through billions of files. As a result, great care must be taken in organizing files and monitoring how data is used. Directories and files may be meticulously named with the *hope* that they can be recalled later or located through basic file browsing. For a storage administrator, not being able to properly find and analyze data can lead to under utilized, poor performing, and less reliable storage systems. For example, if an administrator cannot find out which files are rarely or frequently accessed they cannot determine proper tiering strategies. Ineffective management is perhaps more serious as it puts all data in the file system in jeopardy.

File systems organize files into a hierarchical namespace. This organization provides a number of problems for finding and managing files at large-scales.

1. *Files organization is restrictive.* File system organization closely resembles the way objects would be organized in the physical world, such as files in a filing cabinet. In the physical world, retrieving a file from a filing cabinet may require knowing that it is in first folder, in the second drawer of the third cabinet of the library. Retrieving a digital file requires knowing its physical location in the file system. Retrieving a file may require knowing it is in directory /usr/local/foo/ under the name myfile.txt and under the /cs/students/ mount point. However, the file's location by itself is a poor metric for retrieval because it does not describe the file and is irrelevant to the data that it contains. What matters is being able to find files with whatever information may be known about them.

   Additionally, as in the physical world, files can only be organized along a single axis. For example, academic papers in a physical filing cabinet may be sorted by the author's last name. This organization is helpful if the author's last name is known but useless if only the title or subject of the paper is known. Similarly, in a file system, if directory and file organization is based on the file's owner then it is very difficult to find a file if only its type or subject matter are known. It is not hard to imagine how difficult it would be to find and manage a physical filing cabinet containing billions files; it is similarly difficult to manage file systems of this size. File systems do maintain a number of per-

file attributes, such as size, access and modifications times, extension, and owner that would be useful for retrieval. However, these attributes are not indexed and can only be used for retrieval during a brute force search. Similarly, files can only be related through $parent \rightarrow child$ relationships with directories. Other important inter-file relationships, such as provenance, temporal context, or use in a related project, are lost. Even the physical world is better in some regards. For example, libraries use card catalogs which are a level of indirection between a books location in the library and its attributes (*e. g.*, author, date of publication, topic) that aid retrieval.

2. *File organization and retrieval are manual.* Accessing a file requires manually telling the file system specifically where the file is. When a file's location is not known, the file system's namespace must be manually navigated. The file system provides no way to automatically locate these files or aid the search. For example, if an administrator wants to know which files have not been accessed in the past month then a brute force navigation of the namespace must be done where each file's access times are analyzed. Answering these kinds of questions with brute force search is far too slow to be practical at large-scales.

   Additionally, users and administrators need to answer questions about the properties of the files being stored in order to properly manage their data. At large-scales, it can be very difficult to manually answer these questions because there is often only a limited understanding of what is being stored and how it is being used. For example, answering "which files should be backed up and which should be deleted?", "where are the files with the flight information from my last vacation?", or "which files were recently modified in my source code tree?", are very difficult to answer because they often require traversal of billions of files. At the scale of billions of files, manual organization and access are often not practical.

2.3.1 Search-Based File Management

The data management problem stems from the fact that *file system hierarchies have grown beyond the ability of any single user or administrator to effectively manage them*. The lack of an effective and scalable method to locate and manages files has resulted in the tendency for data utility or usefulness to decrease as the size of the system increases. Fortunately, decades of research and practice in the file systems, information retrieval, and database communities have shown that *search provides a highly scalable retrieval method that can address many of these problems*. File system search improves file system management by allowing files to be retrieved using any of their features or attributes. Retrieving a file requires only knowing *what* one wants rather than *where* to find it. Search eases the burden of organizing and navigating huge file hierarchies and allowing files to be quickly summarized to provide a better understanding of the state of the file system [148]. Additionally, prior work has shown that search is far better aligned with how users think about and recall their data than standard file hierarchies [47, 157, 170]. The scalability of search as a retrieval method is also made evident by it success on the Internet, where search engines, such as Google and Yahoo!, have revolutionized how web pages are organized and accessed. Additionally, file system search has found commercial success on both desktop [14, 66, 110] and small-scale enterprise [55, 67, 85] file systems. There has also been an increasing demand for file search in high-end computing (HEC) [72], cloud [126], personal [142], and enterprise [51] file systems. While we are not suggesting hierarchies are never useful (they are the best solution in some cases), they are a poor choice as a general solution.

2.3.2 File System Search Background

File systems store two kinds of data: the file data itself and metadata, which is data describing the file data. These two kinds of data allow different kinds of file searches to be performed.

2.3.2.1 File metadata

File metadata, such as inode fields (*e. g.*, size, owner, timestamps, *etc.*), generated by the storage system and extended attributes (*e. g.*, document title, retention policy, backup dates, *etc.*), generated by users and applications, is typically represented as $\langle attribute, value \rangle$ pairs that describe file properties. Today's storage systems can contain millions to billions of files and each file can have dozens of metadata attribute-value pairs, resulting in a data set with $10^{10} - 10^{11}$ total pairs. Metadata search involves indexing file metadata such as inode fields and extended attributes.

Metadata search allows point, range, top-k, and aggregation search over file properties, facilitating complex, ad hoc queries about the files being stored. Metadata search can help users and administrators understand the kinds of files being stored, where they are located, how they are used, how they got there (provenance), and where they should belong. For example, it can help an administrator answer "which files can be moved to second tier storage?" or "which application's and user's files are consuming the most space?". Metadata search can also help a user find his or her ten most recently accessed presentations or largest virtual machine images, manage their storage space, or track file changes. Efficiently answering these questions can greatly improve how users and administrators manage files in large-scale file systems. As a result, metadata search tools are becoming more prevalent; recent reports state that 37% of enterprise businesses use such tools and 40% plan to do so in the near future [51]. Additionally, it is one of the research areas deemed "very important" by the high-end computing community [72].

2.3.2.2 File content

File content search involves searching the data that exist within the file's contents. The content that can be searched are keywords, terms, and attributes that are extract from a file using *transducers*. Transducers are programs that read a file's contents and parse specific file types to extract information, which are often string-based keywords. For example, a transducer that can parse `pdf` file types can parse an academic systems paper about file systems and extract

keywords such as $disk$, $log-structured$, or $workload$ from its contents. A large-scale file system may contain thousands of keywords per file, yielding possibly up to $10^{12}-10^{13}$ keyword occurrences.

Content search is the foundation of most modern file system search applications. This type of search is very similar to the type of web search that search engines like Google and Yahoo! provide. These search engines parse keywords and terms from web pages and documents and index them. File content search allows a file to be retrieved using almost any piece of information contained within the file. Search results are ranked using algorithms, such as TF/IDF (term frequencyinverse document frequency) [135], which ranks the files it believes to better match what the user is looking for higher. In contrast, metadata search uses Boolean search where query results either completely satisfies all fields in the query or they do not.

2.3.3 Large-Scale File System Search Use Cases

To further emphasize the importance of search in large-scale file systems we discuss several use case examples.

1. *Managing scientific data.* Large-scale file systems are commonly used for high-performance computation (HPC) applications, such as scientific simulations [39, 178]. A single high-performance physics simulation, for example, can generate thousands of files containing experiment data. The large number of files makes finding those with relevant or interesting results very difficult. As a result, a scientist may go to great lengths to organize the experiment files in a way that aids later navigation [23]. For example, a file's name may be a composition of the experiment's attributes. An experiment that succeeded, took 1 hour and 30 minutes to complete, and calculated a particle collision value of 22 micro-joules may be named `run_1_succ_1h30m_22uj.data`. However, this still requires the scientist to manually parse thousands of files, which is slow, tedious, and inaccurate. On the other hand, file search allows the scientist to easily navigate these files by simply querying the attributes that they are interested in. For example, finding all successful experiment files can be done by querying all `.data` files in the `.../phys_sim/data/`

directory that are associated with the attribute `successful`. Likewise, a scientist may be able to find the average energy from a particle collision by querying all successful tests and averaging the energy attributes of the search results.

2. *Aiding administrator policies.* File system administrators rely on an understanding of what kinds of data are stored on a file system and how it is used to design and enforce their management policies. As file systems grow, it becomes more difficult to gather this information, often resulting in poor management policies. For example, large-scale file systems often employ tiered storage that allows different classes of data to reside on different classes of storage [184]. Finding which files should reside on top-tier storage and which should reside on lower-tier storage is a difficult chore for administrators [111], causing migration policies to often be simplistic and inaccurate. However, search can help an administrator decide on and enforce migration policies. An administrator who wants to migrate all files that have not been accessed in the past six months to tier-two is hard pressed to traverse possibly a billion files to find them. Using search, the administrator can simply query for the files with modification times longer than six months ago and get an answer much faster. Likewise, before deciding on a six-month policy the administrator may try and find how many files are actively being accessed. Without search, finding the files accessed during the course of the day is difficult; however, search requires only a simple daily query to answer this question.

3. *Meeting legal obligations.* With today's increasing digital legislation, such as Sarbanes-Oxley [176] and HIPAA [175], many file systems are *required* to keep data for certain periods of time and to ensure it is not changed and is always available. This is a difficult task in large file systems because it requires an administrator to navigate and monitor up to billions of files to track compliance data. Additionally, the administrator must be able to ensure that these files have not been modified and must be able to produce them when subpoenaed to do so. Failure to meet these requirements can result in potential legal actions. However, search greatly eases these pains as tracking, monitoring, and producing files can all simply be done through queries. For example, an administrator

may attach attributes to a file that define its legal retention period, a hash of its contents for verification, and the internal financial audits it may be related to. Thus, in response to a subpoena or lawsuit, a query on these attributes can quickly bring up all related documents and verify their contents.

4. *Archival data management.* Often the largest file systems are those that act as digital archives [130, 166, 189]. Archival data is often written once and retrieved infrequently, possibly decades later by different users (*e. g.*, children or heirs). As a result, directory and file organizations are often long forgotten and can be too large to easily re-learn. When archived data cannot effectively be retrieved or cannot later be found, it makes the preservation of the bits less important. However, search provides a simple and effective mechanism for the contents of a digital archive data to later be found. For example, if one were to inherit their grandparent's archived data in a will, and were looking for digital photos from a specific event, they could easily find this data through by query attributes related to the event. Likewise, if an archive stores medical records, a simple search for the patient and the date can return their records. Manual search of an archive may take weeks and not guarantee that the data is found.

5. *Everyday file usage.* While the examples above demonstrate areas where search is helpful, they do not represent most users' everyday interactions with the file system. Typically, large-scale file systems are used for a variety of applications, with different users performing different tasks. For example, a large file system may have one user managing digital photos taken during an archeology excavation, one using AutoCAD drafting files, others working on a source code project, and others working on financial documents. Each of these users faces the challenge of organizing and managing their data. File search can ease the burden of trying to organize files into hierarchies because users are no longer worried about forgetting a single file path. Likewise, they no longer need to waste time or risk losing files when they cannot recall their pathnames. Since organizing, finding, and managing files is the primary way users interact with the file system, search has the potential to drastically change how the file system, in general, is used and to improve its

overall utility. In other words, it can potentially revolutionize the file system as it did with the Internet.

2.4 Large-Scale File System Search Challenges

While search is important to the overall utility of large-scale file systems, there are a number of challenges that must be still be addressed. Both discussions with large-scale file system customers [129] and personal experience have shown that existing enterprise search tools [12, 55, 67, 85, 109] are often too expensive, slow, and cumbersome to be effective in large-scale systems. We now discuss some of the key challenges.

1. *Cost.* Searching large-scale file systems requires indexing billions of attributes and keywords. Large-scale search engines and database systems rely on dedicated hardware to achieve high-performance at these scales. Dedicated hardware allows these systems to serve searches from main-memory, utilize all CPU resources, and not worry about disk space utilization. Additionally, most are not embedded within the system, requiring additional network bandwidth to communicate with the storage server. Businesses that use these search systems, such as large banks and web search companies, can afford to provide this hardware because search performance is key to their business's success. Many DBMSs used in banking systems assume abundant CPU, memory, and disk resources [75]. Large-scale search engines, such as Google and Yahoo! use large, dedicated clusters with thousands of machines to achieve high-performance [15]. However, with these hardware requirements it can cost tens of thousands of dollars to search just millions of files [65]. This cost is far too high for most file system budgets. Even if the cost can be budgeted, other file system necessities, such as I/O performance and capacity, generally take precedence. As a result, reliance on dedicated hardware for high-performance search can limit deployment in most large-scale file systems.

2. *Performance.* Search performance is critical for usability. For example, web search engines aim to return results in several hundred milliseconds [41]. Likewise, update

performance is important because updates must be frequently applied so that search results accurately reflect the state of the file system. Achieving fast search and update performance is difficult to do in file systems with billions of files and frequent file modifications [10, 48, 178]. Unfortunately, current solutions often rely on general-purpose, off-the-shelf indexing and search technology that is not optimized for file systems. Although standard indexing solutions, such as DBMSs have benefited from decades of performance research and optimizations, such as vertical partitioning [87] and materialized views, their designs are not a perfect fit for file system search. These systems lack file system specific optimizations and have functionality that is not needed for file system search and which can add overhead even when disabled [165]. This is not a new concept; the DBMS community has argued that general-purpose DBMSs are not a "one size fits all solution" [28, 162, 165], instead saying that application-specific designs are often best. Similarly, Rosenblum and Ousterhout argued that "file system design is governed by...technology...and workload" [138], which is *not* the approach taken by general-purpose solutions. As a result, it is difficult to achieve scalable, high-performance search in large-scale file systems. While many desktop search systems can achieve performance on a single, small desktop, it is difficult to scale these solutions to multi-user file systems with billions of files.

3. *Data collection and consistency.* File search is often implemented as an application outside of the file system. In order for search results to accurately reflect the file system, file changes must be extracted from the file system and these changes must be replicated in the search application's index. However, large-scale file systems have billions of files and highly demanding workloads with rapidly changing files [10, 48, 178]. This size and workload make efficiently collecting these changes very difficult. Also, collection methods such as crawling the file system or interposing hooks along I/O paths, can have a negative impact on file system performance. We have observed commercial systems that took 22 hours to crawl 500 GB and 10 days to crawl 10 TB. Due to the inability to quickly

collect changes and update the index, large-scale web search engines updates are applied off-line on a separate cluster and the index is re-built weekly [15].

4. *Highly distributed.* In order to achieve needed performance and scalability, large file systems are distributed. Similarly, a large-scale search index, which can have space requirements that are as high as 20% of the file system's capacity [20], must also be distributed. However, distributed search engines, such as web search engines, often perform static partitioning across a cluster of dedicated machines and perform only manual, off-fline updates [106, 132]. These hardware and update methods are not feasible for file system search, which must often be integrated with the file system to limit cost, must be frequently updated, and handle changing workloads. Additionally, distributing the index can help to co-locate indexes near the files that they need to access and provides parallel execution, such as with MapReduce [40].

5. *Ranking.* Searching the web has been greatly improved through successful search result ranking algorithms [29]. These algorithms often rank results so well that they only need to return the few top-K results to satisfy most queries [151]. However, such algorithms do not yet exist for file systems, particularly, large-scale file systems. Current desktop and enterprise file systems often rely on simplistic ranking algorithms that require users to sift possibly through thousands of search results. File systems currently lack the semantic information, such as hyperlinks, that web rankings algorithms leverage [25]. In large-scale file systems, a single search can return millions of results, making accurate ranking critical. To address this problem there has been an increasing amount of work looking at how to use semantic links [73, 149, 155] in the file system can improve ranking.

6. *Security.* Large-scale file systems often store data, such as nuclear test results, that make security critical. File system search should not leak privileged data otherwise it cannot be used in a wide variety of systems. Unfortunately, current file system search tools either do not enforce file permissions [12] or significantly degrade performance [33] to do so. In many cases, security is addressed by building a separate index for each user [33]. This approach guarantees that the user has permission to access the files in his or her

index. However, this approach also requires prohibitively high disk space since files are often replicated in many indexes. Moreover, changing a file's permissions can require updating a large number of user indexes. Another approach is to perform permission checks (*i. e.*, `stat()` calls) for every search result and only return results that pass the check. However, performing a permission check on what may be millions of files can significantly impact performance and pollute file system caches.

7. *Interface.* The traditional file system interface (*i. e.*, POSIX) has lasted more than three decades in part because it uses a simple (albeit limited) organization paradigm. However, search-based access methods require a more complicated interface since files can be accessed with any of their attributes, search results must be shown, and inter-file relationships need to be visualized. Moreover, the interface must be simple enough for users to be able and willing to use frequently. Basic search interfaces, such as the simple Google keyword box, are likely too simplistic to express the kinds of queries that users need to ask, while languages such as SQL [35] are likely too complicated. Current and previous work has looked at how to use new interfaces [7, 89] to improve interaction with the file system.

2.5 Existing Solutions

This thesis builds on work in the file system, information retrieval, and database communities. We now highlight existing work in file system search from these fields and discuss the challenges that still remain for large-scale file system search.

2.5.1 Brute Force Search

Early file system search tools aimed to make brute force search less cumbersome. Tools such as `find` and `grep` walked the file system hierarchy, accessing each file, and checking whether it matched the query. When file systems were relatively small these tools were quite effective. However, at large-scales this approach is not practical because searches can take hours or days and utilize most of the file systems resources.

A number of approaches have looked at how to improve the performance of brute force search. For example, MapReduce [40] can distribute `grep` operations across a cluster so that they are close to the data they need to access and run in parallel. Diamond [81] uses an approach called *Early Discard* to quickly identify if a file is relevant to a query. Early Discard used application-specific "searchlets" to determine when a file is irrelevant to a given query. This approach reduces the amount of data that must be read by analyzing a small part of a file to determine whether it is worth continuing to search.

2.5.2 Desktop and Enterprise Search

More recently search systems have relied on indexing to improve performance. The index is a data structure that pre-computes the location of attributes and keywords in the file system. Thus a query only needs to perform an index look up rather than a traversal of the entire file system. This is the approach taken by the many of desktop [12, 66, 108, 110] and small-scale enterprise [55, 67, 82, 85] file system search applications. These applications often consist of a general-purpose relational database (DBMS) and an inverted index. The DBMS provides structured metadata search while the inverted index provides unstructured content search. Additionally, they are implemented as applications outside of the file system. These applications supplement the file system's lack of search support and do not require the file system to be modified. The file system is periodically crawled to reflect new file changes in the search application's index.

Virtually all major operating systems (Windows, OSX, and Linux) now ship with file system search functionality included. Additionally, recent reports show that most enterprise businesses use or are planning on using an enterprise file system search appliance in the near future [51]. Businesses often use these appliances to improve employee productivity and to ensure legal compliance guidelines are followed. Both desktop and enterprise search target smaller-scale systems, storing gigabytes of data and millions of files [65]. As a result, the general-purpose indexing solutions these applications use are relatively effective at such scales. Since these applications reside outside of the file system, file attributes are usually collected by

walking the file system namespace and reading each file's metadata and data. Often *transducers* are used to parse various file type formats (*e. g.*, `doc`, `pdf`, *etc.*) and extract keywords.

While desktop and enterprise file system search is popular and becoming ubiquitous, it is not well suited to scale to large-scale file systems in either cost, performance, or manageability. Reliance on general-purpose index designs make few file system optimizations, do not efficiently enforce security, and require significant file system CPU, memory, and disk resources (for example, enterprise appliances ship with their own dedicated hardware resources [65]). Additionally, crawling the file system and updating the index are very slow which causes the search index and file system to be frequently inconsistent which can yield incorrect search results. Thus, these systems serve to demonstrate the importance and popularity of file system search but also demonstrate the challenges in scaling to very large file systems.

2.5.3 Semantic File Systems

File system search applications, such as desktop and enterprise search tools, do not address the limitations of hierarchical file systems. Instead they provide an auxiliary application that can be searched. However, the limitations of hierarchical file system organizations, which were identified over two decades ago [113], have prompted significant research into how files systems should present information to users. One of the first file systems to do this was the Semantic File System (SFS) [63]. SFS points out the limitations of traditional file system hierarchies and proposes an extension to the hierarchy that allows users to navigate the file system by searching file attributes that are automatically extracted from files. Virtual directories, which are directories whose contents are dynamically created by a query, are used to support legacy applications. These file systems are called "semantic" because the namespace allows files to be organized based on their attributes and semantic meaning (*e. g.*, with virtual directories) rather than simply a location. Thus, a user can navigate a dynamically created hierarchy based on any file attributes. The SFS design is made to work as a back-end file server with existing NFS and AFS protocol, however, it is not distributed across multiple servers. SFS relies on B-trees to index and provide search over file attributes.

SFS introduced the concept of a dynamic, search-based namespace for file systems. A number of file systems extended the basic concepts described in SFS to provide additional functionality. The Hierarchy and Content (HAC) [69] file system aims to integrate existing hierarchical file organizations with content-based search of files, with the goal of lowering the barrier to entry to semantic file systems for users. The authors argued that a key reason that SFS never caught on as a popular file system design was that it was too difficult for users to move from a standard hierarchical file system to a search-based one. Starting with a traditional hierarchical file system, HAC adds content-based access by allowing new semantic directories to be created, which are normal directories that are associated with a query and contain symbolic links to search results and can be scoped to only a part of the hierarchy. They introduced the notion of *scope consistency* to deal with file attribute changes in nested virtual directories. HAC was implemented on top of a normal UNIX file system and uses the GLIMPSE [104] file system search tool provide search of file contents.

The pStore file system [188] extends traditional file systems to allow complex semantic file metadata and relationships to be defined. They introduce a new model for defining the attributes of a file and data schemas that are more flexible than those offered by traditional databases. Their data schemas are based on the Resource Description Framework (RDF). These schema's are used to construct customized namespaces that are meaningful to specific users or applications. They also argue that general-purpose databases are not the right for storing structures for semantic file system data.

The Logic File System (LISFS) [124] take a more theoretical approach to semantic file retrieval. LISFS propose a new model where file names are represented as Boolean equations. For example, accessing a file can be done with a Boolean equation in conjunctive normal form such as $(a_1 \vee a_2) \wedge (\neg a_3)$, where a_1, a_2, and a_3 are file attributes. Using these equations, file paths can be constructed that allow a large number of different paths to be used to locate a file. In addition, traditional file systems hierarchies can be constructed because these equations exhibit a commutative property. File attributes are stored in special tables that implement directed acyclic graphs and file retrievals are translated in to axiomatic look-ups in these tables.

The Property List DIRectory system (PLDIR) [113] provided better ways to internally represent file attributes in a file system. PLDIR defined a general model for describing metadata using property lists, which could be used to represent file search abstractions. While its indexing capabilities were very basic, it did address some index update and consistency issues.

Damasc is a file system intended for high-performance computing environments [27]. Damasc provides a declarative query interface to files, which is effective for many common HPC queries. File data is stored in a normal file system with a number of modules above it that provide file parsing, indexing, provenance collection, and interface capabilities. Indexes are constructed based on application query patterns in order to speed up common searches.

Other semantic file systems provide better file retrieval mechanisms by improving naming rather than adding explicit file search. For example, the Prospero file system [117] builds a global file system where files are scattered across multiple machines and each user builds there own view of the data on the system, rather than trying to organize a global namespace. In that way, no matter how large the file system is, users only see and only deal with the files that they deem relevant to them. The Linking File System (LiFS) [6] is designed to express semantic links between files (*e. g.*, provenance and context) through actual links implemented in the file system. This allows users to use these links for quick traversal of the file system along axes other than just the hierarchy.

Semantic file systems provide a more scalable way to organize, navigate, and search files than traditional file system hierarchies. However, most do not focus on the underlying search algorithms and data structures that enable scalable search performance. Instead, they focus on the interface and query language and use basic general-purpose indexes, such as simple B-trees or databases. Additionally, most are implemented on top of a traditional file system with a number of separate search applications that are used to index file attributes. Thus, while they provide a more scalable file system interface, they do not provide scalable search performance or consistency guarantees that are needed to completely address search problems in massive-scale file systems. At relatively small scales (*e. g.*, tens of thousands of files), file access performance can still be $3 - 5\times$ slower than in a traditional file system.

2.5.4 Searchable File Systems

There are other file systems that enable file search without foregoing the traditional hierarchical file systems. The Be File System (BeFS) [61] has support for file search built directly into the file system. BeFS stores extended attributes in hidden file system directories that are indexed in B+trees, allowing queries to be issued over them. These B+trees are stored in a special directory with each attribute being represented as a directory entry and the index being stored as a file pointed to by the entry. Common attributes, like file size, owner and modification times are automatically indexed by BeFS and the user can specify any additional attributes to be indexed.

The Inversion File Systems [119] takes a different approach than BeFS. Inversion implements the file system using a PostgreSQL [163] database as the back-end storage mechanism. This provides transactional storage, crash recovery, and the ability to issue database queries over the file system. Each file is represented as a row in a table. File metadata and data are stored in separate tables where the metadata table points to file data in the data table. Each normal file system operation maps to an SQL query.

The Provenance-Aware Storage System (PASS) [114] stores provenance (*i. e.*, lineage) metadata with each file, showing where its data came from and how it was created. This tackles a specific set of queries users and administrators often have about their data, such as, "which source files were used to build this binary?" or "which file was this paragraph copied from?". All provenance attributes are stored in Berkeley DB databases [120] and a query layer is added on top of these databases to allow users to search provenance data. Most of the underlying file system is left unchanged except for the operations that are annotated to ensure provenance is kept.

2.5.5 Ranking Search Results

In addition to making search fast and consistent with the file system, another key challenge is making searches effective. That is, searches are actually able to help users find the

data they are looking for. The importance of effective search is evident on the Internet, where search engines can satisfy searches with only the top few (*e. g.*, ten) ranked search results [151].

Web ranking algorithms, such as Google's PageRank [29], benefit from the semantic links between pages, which it can use to infer the importance of a page. As a result, most file system ranking algorithms attempt to extract semantic information from files to improve ranking. The Eureka [25] ranking algorithm extracts semantic links from files, such as name overlap or shared references, that allows techniques similar to PageRank to be used for file systems. Similarly, Connections [155] extracts temporal relationships from files, such as two source code files being opened at the same time, that are used as semantic links. Connections builds a graph of these links that is used to reorder search results from a file system's search engine. Shah, *et al.* [149] infer semantic links using file provenance and build a graph-based ranking method similar to that of Connections. They conducted a user study to validate that such ranking improvements do enhance the overall user search experience. Finally, Gyllstrom, *et al.* [73] only consider the data that users have actually seen (*i. e.*, information displayed on the screen) as searchable information. This approach is built around the intuition that users are only going to search for information that they have seen previously and that if they never saw it, and therefore likely do not know it exists, it is unlikely they will want to search for it. This approach reduces the overall size of the search corpus and puts more weight on ranking data that the user has seen before. Additionally, web search has shown that personalized ranking and views of data can greatly enhance search relevance [83, 90, 99]. This approach has the potential to be useful on large-scale file systems where search needs, and often the files to search, vary between users.

2.6 Index Design

Fast search of a file system is achieved by building an *index* before hand. The two fundamental index structures used in file system search are relational databases (DBMS) [37] and inverted indexes [74, 192]. DBMSs are often used to store file metadata, which is structured $\langle attribute, value \rangle$ pairs such as inode fields and extended attributes. This structure maps well

onto the DBMS relational model. Inverted indexes are used to store unstructured file keywords, which are often text-based. Inverted indexes, which are often called text databases, are the primary method of text indexing. These two search structures have been researched for decades and have a considerable number of different designs. We now highlight those related to file system search.

2.6.1 Relational Databases

The traditional relational database architecture is rooted in the design of Systems R [16], which was designed more than 25 years ago. System R and its descendants, such as DB2, MySQL, and PostgreSQL, are designed for business data processing workloads. At the time, these systems were designed to provide indexing and storage structures that were disk-oriented, relied heavily on threading for performance, coarse locking mechanisms for transaction support, and log-based recovery.

In most DBMSs, data is stored in a table with a generally fixed number columns and variable number of rows that define relations. For example, a table for a file system may contain an inode number, owner, size, file type, and modification time as columns. As more files are added, more rows are added to the table. Each row is stored sequentially on-disk, making it efficient to read and write an entire row. New rows are appended to regions of the table, making it efficient to write a large number of consecutive rows.

These DBMSs are often called row-stores or write-optimized databases since they are optimized for reading and writing entire rows and are efficient for writing new rows since they are appended sequentially to the table. Additionally, this approach is space efficient since only approximately $N \times M$ bytes are used, for N rows each with M columns. In the simplest case, querying the table involves a full sequential table scan to find the rows that match the query. When data is clustered in the table or when most of the table must be read, this can be quite efficient. However, when this is not the case, table scans can be extremely slow as extra data may be read and a large number of disk seeks may be incurred.

There are a number of optimizations that can improve query performance. A common approach to improving query performance is building B+-tree indexes on the columns in the ta-

ble that need to be quickly searched. Each B+-tree is indexed on a column's attribute values and stores lists of row IDs that match the value. Queries for a specific attribute value or range of values can quickly identify the specific rows in the table that need to be retrieved, avoiding the cost of a table scan. This approach does not, however, eliminate the seeks that may occur if the rows are not adjacent in the table. When multiple indexed attributes are queried, such as querying for files where $(owner = Andrew) \wedge (size > 5\,MB)$, a *query planner* chooses which attribute it believes to be most selective (*i. e.*, the attribute that will yield the fewest search results) and searches that attribute's index. The N results from that index are then linearly scanned and filtered to make sure that the final results satisfy both query predicates. This pruning will thus yield between $[0, N]$ results. By selecting the index with the fewest likely results, the fewest results need to be linearly scanned.

Selecting records from the index likely to have the fewest results is a much faster approach than searching both indexes and calculating the union of their results since less data is read from the table and processed. However, doing so requires the query planner to be able to efficiently chose which index will produce the fewest results. To do this, query planners use selectivity estimators to estimate the selectivity of predicates in a query. More specifically, it estimates the number of tuples that will satisfy the predicate. It uses the statistical information about the table that the DBMS maintains. Unfortunately, when attribute value distributions are skewed, query planners tend to be inaccurate [101]. This is because the sampling methods, which rely on only a subset of the data and must be general, have difficulty adapting to the variance in highly skewed distributions. Additionally, when both values match many records, extra data will still be read and is processed, even if their intersection is small. It should also be noted that building additional data structures, such as B+-tree indexes, does incur a space overhead. This overhead grows proportionally with the size of the table and increases as more columns are indexed. Additionally, update performance decreases because, in addition to the table, each index must also be updated.

Another approach to improving performance is vertical partitioning [87]. Vertical partitioning physically breaks the table up into multiple tables, one for each column. Each table stores an additional position integer that allows rows to be identified across tables. Often tables

are sorted to provide more efficient scanning. Many queries, such as in the example above, do not need to retrieve an entire row's worth of data. In a non-partitioned table, a scan requires retrieving the full row for each row scanned, which can be very slow. By partitioning the table along column axes, only columns with data relevant to a query need to be retrieved. Joins are then used (often hash joins) to facilitate searches across columns. Compared to building an index on each column in a non-partitioned table, vertical partitioning can be faster in cases where only a few columns are needed from many rows. This is because less disk bandwidth is used as the row data that is not used in the query is not read. However, vertical partitioning is a trade-off with computation overhead as join operations must be used across the tables and can impede performance [1]. Also, storing an extra position ID with each row in each table to identify which tuple it belongs to adds a space overhead, potentially doubling the space required if all other attributes are integers. Vertical partitioning also degrades update performance since writing a row now requires a disk seek between each table.

Column-stores are a more recent DBMS design that are read-optimized, rather than write optimized like traditional DBMSs [164]. A column-store DBMS stores a table's column data contiguously on-disk instead of row data. Doing so allows queries that need to read only a subset of the columns to perform much faster than if the entire row had to be read. Additionally, many column-stores use compression and dense packing of column values on-disk to reduce disk bandwidth requirements [56, 164] and lean more heavily on CPU performance. Additionally, extra tuple information does not need to be stored with each column. Column-stores use join indexes to provide more efficient queries across columns. However, since column-stores are read-optimized, their designs comes at a cost to write performance. Writing an entire row now requires a disk seek between each column, similar to vertical partitioning, which decreases performance.

The column-store design stems from significant changes in technology trends and business processing workloads since the 1970s when the original System R was designed. Processor speeds have increased significantly as have disk capacities, while disk bandwidth has increased at a much slower rate. Additionally, the price per gigabyte of memory has decreased [165]. These trends make the heavily disk-based DBMS data structures a bottleneck.

Similarly, business processing workloads (*e. g.*, OLTP, OLAP, and data warehousing) have become more read heavy and their need for heavyweight logging, threading, and transactional operations have been decreasing [75].

These continual changes in technology and workload, and a continued reliance on traditional general-purpose DBMSs, have given rise to a belief that existing DBMS designs are not a "one size fits all" solution [28, 162, 165]. That is, a general-purpose DBMS cannot simply be tuned and calibrated to properly fit every workload. Many in the database community have argued and shown [75, 161] that using a traditional DBMS for a variety of search and indexing applications is often a poor solution and that a customized, application-specific design that considers the technology and workload requirements of the specific problem can often significantly outperform general-purpose DBMSs.

The "one size fits all" concept suggests that general-purpose DBMSs may not be an optimal solution for file system search. DBMSs were designed for business processing in the 1970s, not modern large-scale file system search. Achieving reasonable performance will often require additional hardware as well as a database administrator to constantly "tune" database knobs. DBMSs also have functionality, such as transactions and coarse locking, that are not needed for file system search and can add overhead [165]. Similarly, as evidenced by their contrasting designs, databases are often optimized for either read or write workloads and have difficulty doing both well [1, 76, 78]. File system search must provide both fast search performance and frequent, real-time updates of the index.

2.6.2 Inverted Indexes

Inverted indexes [74, 192] are designed to be text databases and are the foundation of content (also known as keyword full-text, or term) search on the Internet and in current file system search. An inverted index, for a given text collection, builds a *dictionary* that contains a mapping for each of the K keywords in the collection to a list of the locations where the keywords occur in the file system. The basic inverted index architecture is illustrated in Figure 2.3. Each keyword location is known as a *posting* and the list of locations for a keyword is called the keyword's *posting list*. For a file system, these locations are generally offsets within files

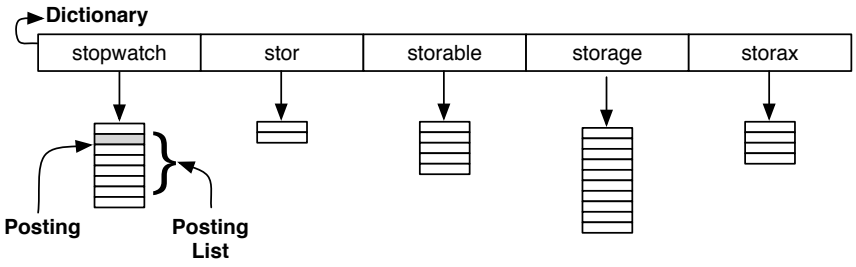


Figure 2.3: **Inverted index architecture.** The dictionary is a data structure that maps keywords to posting lists. Posting lists contain postings that describe each occurrence of the keyword in the file system. Searches require retrieving the posting lists for each keyword in the query.

where the keyword occurs. In most cases, each posting in the list contains a tuple with the ID of the file (such as an inode number) in which the term occurs, the number of times its occurs, and a list of offset locations within the file for each occurrence. Thus each posting generally has the format $(docid, tf, \langle p_1, ..., p_{tf} \rangle)$, where $docid$ is the unique document identifier, tf is the number of term occurrences, and p_i is the offset within the file where the term occurs.

The dictionary is usually implemented as a hash table or sort-based data structure that allows efficient look up using a single term as a key. Dictionary size can often grow quite large since it requires close to $K \times N$ bytes, where K is the average keyword length in bytes and N is the number of keywords. For large-scale file systems with many keywords, even with compression, the dictionary will be far too large to fit in a single machine's main memory [30]. Thus, they are often either stored on-disk or distributed across a cluster of machines.

Each posting list is stored sequentially on-disk. Given a query, such as $index \wedge search \wedge storage$, each keyword is looked up in the dictionary and their corresponding posting lists are read from disk. The union of the posting lists are calculated and a result ranking function is then applied to produce the final search results. In most cases, search performance is a function of the amount of data that must be read from disk, the number of disk seeks incurred to read the data, and the amount of processing (*e. g.*, decompression, ranking, file permission checks) that must be done to produce the final set of results.

However, keeping posting lists sequential on-disk can make list updates very slow, especially as the posting list grows in size. As a result, posting list incremental update algorithms are generally either optimized for search performance (*i. e.*, try and maintain posting list sequentiality) or update performance (*i. e.*, sacrifice posting list sequentiality for fast writes) [96]. *In-place update* algorithms are update optimized approaches. When a posting list is created, a sequential region on disk is allocated that is larger than the required size, leaving some unused space at the end. As new postings are added, they are written into the over allocated regions of the posting list, without requiring any extra data reads or disk seeks. If the over allocated region is not used it becomes wasted space. If the posting list grows beyond the size of the region, a new sequential region is allocated elsewhere on disk to place new postings. A disk seek is then required to read the entire posting list (*i. e.*, all on disk regions). Thus, as posting lists grow in size, in-place updates can make search performance poor as many disk seeks may be required to read a posting list from disk. *Merge-based* algorithms are search optimized approaches. When a posting list is created, a sequential region of the exact size is allocated on disk. When the list is modified, the original list is read from disk, modified in-memory, and then written sequentially to a new location on disk. Thus, merge-based approaches incur no extra space overhead and ensure that posting lists are sequential on disk. However, update performance can be very slow since the entire posting list must be read and written for each update.

However, there has been additional work to improve the trade-offs between search and update performance. Hybrid index maintenance [32] divides the index into two sections, one that is managed with in-place updates and one that is managed with merge-based updates. This division is based on posting list length where short posting lists (*i. e.*, shorter than some calculated threshold V) are managed with merge-based updates. Short posting lists can more easily be read and written with a merge-based update and would waste significant disk space under an in-place approach. Longer posting lists are then managed with an in-place strategy where merge-based updates would simply be too slow. However, an in-place update mechanism still requires extra disk seeks to retrieve these long posting lists that are larger than a single region. Often there is a delay between when file changes are collected and when they are reflected in the index. The JiTI [93] algorithm allows changes that have not yet been applied to

the index and are buffered in-memory to still be searched. A postings cache prepares posting lists in-memory prior to being written out to the main index. A separate query engine then issues queries over both the main index and the postings cache, allowing the most recent updates to also be returned.

Web search engines, such a Google and Yahoo!, maintain possibly the largest inverted indexes, indexing trillions of web pages [68]. These search engines utilize multi-million dollar data centers to search for this many pages. The inverted index is partitioned across the data center using one of two methods: the *local inverted file* (IF_L) or the *global inverted file* (IF_G) [15, 132, 133]. The IF_L strategy divides posting lists across the P servers in the data center. Searches are then broadcast to all P servers and each returns a disjoint set of results. The IF_G strategy partitions the keyword space up amongst the P servers, where each server is responsible for a part of the space, such as $1/P$ fraction of the keywords. Each of the P servers has enough memory and proxy cache nodes to ensure that queries can be served without going to disk. The index is re-built *off-line* on a close to weekly basis after additional web pages are crawled [15]. During this re-build process, a large-scale staging area (a separate cluster of machines) is used to modify or re-create posting lists and calculate page rankings. An additional link index is often used to improve rank calculations.

While inverted indexes are the mainstay of modern text retrieval, not all of the designs are well suited for file system search. Small-scale, disk-based inverted indexes make significant trade-offs between search and update performance. As a result, either updates are too slow to handle real-time update requirements or search performance becomes unacceptably slow. Additionally, small-scale designs have few methods for appropriately distributing the index. Large-scale designs, such as web search engines, are far too expensive and heavyweight to be effective for a file system. They require dedicated resources and re-build the index too infrequently. These issues suggest that designs that are tailored to file system search may able to improve performance.

2.6.3 File System Optimized Indexing

We are not the first to argue that general-purpose indexing solutions are not the right approach to file system search. As a result, there are a number of file system specific indexing techniques that exist. GLIMPSE [104] reduced disk space requirements, compared to a normal full-text inverted index, by maintaining only a partial inverted index that does not store the location of every term occurrence. They argued that the space requirements for maintaining a full inverted index were too high for most file systems. GLIMPSE partitioned the search space, using fixed size blocks of the file space, which were then referenced by the partial inverted index. A tool similar to `grep` was used to find exact term locations with each fixed size block. However, the GLIMPSE tool is almost two decades old and disk capacity is not nearly as significant a problem today as it was then. As a result, in most cases, even for file systems, maintaining a full inverted index for performance is preferred.

Geometric partitioning [95] aimed to improve inverted index update performance by breaking up the inverted index's inverted lists by update time. The most recently updated inverted lists were kept small and sequential, allowing future updates to be applied quickly. A merging algorithm created new partitions as the lists grow over time. Query-based partitioning [112] used a similar approach, though it partitioned the inverted index based on file search frequency, allowing index data for infrequently searched files to be offloaded to second-tier storage to improve cost. The Wumpus desktop search system [31] introduces a number of improvements to conventional inverted index design, which improves full-text search on desktop file systems. However, its current design targets desktop file systems and lacks a number of features critical to large-scale file system search. SmartStore [80] is a search system that indexes metadata in a distributed R-tree and uses Latent Semantic Indexing (LSI) to group correlated metadata. SmartStore does not handle real-time updates, and the use of LSI limits its ability to perform index updates quickly.

2.7 Summary

There is a rapid increase in the amount of digital data being produced and it is only expected to increase in the future. In this chapter we described how this rapid growth in data volumes has introduced a new challenge for large-scale file systems: effectively organizing, finding, and managing such a large sea of information. In response to this challenge there has been an increasing trend towards search-based file access. Through a discussion of file search properties and use cases we showed how search addresses many of the limitation and restrictions of current simple hierarchical file organizations.

We argued that existing file system search solutions are ill-equipped to scale to billions of files. We discussed the search challenges that make existing solutions to too expensive, slow, and cumbersome to be effective at such scales. We showed that these systems rely on general-purpose index designs that are not designed for file system search and can limit search and update performance. We also provided a high-level overview of how general-purpose indexes such as DBMSs are used for file system search. Additionally, we described how designing search systems outside of the file system causes consistency and efficiency issues that are difficult to address.

These observations drive the motivation for the following chapters. Since effective file system design is driven by an understanding of the kinds of data stored and how it is used, we begin by looking at the properties of large-scale file systems. We then explore how the use of general-purpose index structures can be avoided with index designs that are designed specifically for file systems and which leverage file system properties. We use these new indexing techniques to explore how search functionality can be integrated directly into the file system to avoid the need to maintain a replication of all file attributes in a separate search application.

Chapter 3

Properties of Large-Scale File Systems

Large-scale network file systems are playing an increasingly important role in today's data storage. The motivation to centralize data behind network file systems has been driven by the desire to lower management costs and the need to reliably access growing amounts of data from multiple locations and is made possible by improvements in processing and network capabilities. The design of these systems [88, 116, 138] is guided by an understanding of the kinds of data being stored and how data is being used, which is often obtained by measuring and analyzing file system traces.

There are two different methods used to understand file system properties; measuring and analyzing workloads, which shows how data is being used as well as measuring and analyzing snapshots, which shows the kinds of data being stored. Workload traces contain data access requests, such as a file being opened and read. Snapshot traces contain information about the properties of files stored at a given time, such as their location in the namespace and the types of files (*e. g.*, `doc`, `pdf`).

While a number of trace-based file system workload [17, 48, 123, 137, 177, 178] and snapshot [5, 22, 39, 43, 145] studies have been conducted in the past, there are factors indicating that further study is necessary. First, the last major network file system trace study [48] analyzed traces from 2001, over half a decade ago; there have been significant changes in the architecture and use of network storage since then. Second, no published workload or snapshot study has ever analyzed large-scale enterprise network file systems, focusing instead on research or desk-

top systems, such as those seen in university settings. While useful, their findings likely differ from those in large enterprise file systems. Third, most studies focus on analyzing properties that are applicable to improving disk performance. There is little analysis that focuses on file organization and workload semantics, which are important for designing better organization and indexing strategies.

In this chapter, we collect and analyze traces of *both* file system workloads and snapshots from real world, large-scale enterprise network file systems. Our workload traces were collected over three months from two network file servers deployed in NetApp's data center. One server hosts data for the marketing, sales, and finance departments, and the other hosts data for the engineering departments. Combined, these systems contain over 22 TB of actively used storage and are used by over 1500 employees. We traced CIFS [92] network traffic, which is the primary network file protocol used in Windows. Our snapshot traces were collected from three network file servers also deployed at NetApp. One server hosted web and Wiki server files, another was a build server for the engineering department, while another served employee home directories. Combined, these systems contain almost 80 TB of actively used storage. The analysis of these traces focused on: (1) changes in file access patterns since previous studies, (2) aspects specific to network systems, such as file sharing, and (3) how the namespace impacts file attributes. All of which are important for better understanding how we can organize and manage files.

Our analysis revealed important changes in several aspects of file system workloads. For example, we found that read-write file access patterns, which are highly random, are much more common relative to read-only and write-only access patterns as compared to past studies. We also found our workloads to be more write-oriented than those previously studied, with only about twice as many bytes read as written. Both of these findings challenge traditionally held assumptions about access patterns and sequentiality. We also found that metadata attribute values are highly clustered in the namespace and have very skewed distributions. These findings demonstrate important properties about how files are organized in large-scale network file systems. A summary of key observations can be found in Table 3.1.

Compared to Previous Studies
1. Both workloads are more write-oriented. Read to write byte ratios have signifcantly decreased.
2. Read-write access patterns have increased 30-fold relative to read-only and write-only access patterns.
3. Most bytes are transferred in longer sequential runs. These runs are an order of magnitude larger.
4. Most bytes transferred are from larger files. File sizes are up to an order of magnitude larger.
5. Files live an order of magnitude longer. Fewer than 50% are deleted within a day of creation.
New Observations
6. Files are rarely re-opened. Over 66% are re-opened once and 95% fewer than five times.
7. Files re-opens are temporally related. Over 60% of re-opens occur within a minute of the first.
8. A small fraction of clients account for a large fraction of file activity. Fewer than 1% of clients account for 50% of file requests.
9. Files are infrequently shared by more than one client. Over 76% of files are never opened by more than one client.
10. File sharing is rarely concurrent and sharing is usually read-only. Only 5% of files opened by multiple clients are concurrent and 90% of sharing is read-only.
11. Most file types do not have a common access pattern.
12. Specific file metadata attribute values exist in only a small fraction of directories.
13. A small number of metadata attribute values account for a large fraction of the total values.

Table 3.1: **Summary of observations.** A summary of important file system trace observations from our trace analysis. Note that we define clients to be unique IP addresses, as described in Section 3.3.1.

The remainder of this chapter is organized as follows. The previous workload studies are discussed in Section 3.1 and our workload tracing methodology is discussed in Section 3.2. Our workload analysis is discussed in Section 3.3 with the implications described in Section 3.4. Section 3.5 describes the previous snapshot studies. How we collected the snapshot traces is detailed in Section 3.6. Our snapshot trace observations and their impact are presented in Section 3.7. In Section 3.8 we summarize this chapter.

3.1 Previous Workload Studies

File system workload studies have guided file system designs for decades. In this section, we discuss the previous studies, which are summarized in Table 3.2, and outline factors that we believe motivate the need for new file system analysis. In addition, we provide a brief background of the CIFS protocol.

Study	Date of Traces	FS/Protocol	Network FS	Workload
Ousterhout, *et al.* [123]	1985	BSD		Engineering
Ramakrishnan, *et al.* [131]	1988-89	VAX/VMS	x	Engineering, HPC, Corporate
Baker, *et al.* [17]	1991	Sprite	x	Engineering
Gribble, *et al.* [71]	1991-97	Sprite, NFS, VxFS	x	Engineering, Backup
Vogels [177]	1998	FAT, NTFS		Engineering, HPC
Zhou and Smith [191]	1999	VFAT		PC
Roselli, *et al.* [137]	1997-00	VxFS, NTFS		Engineering, Server
Ellard, *et al.* [48]	2001	NFS	x	Engineering, Email
Anderson [10]	2003 & 2007	NFS	x	Computer Animation
Leung, *et al.*	2007	CIFS	x	Corporate, Engineering

Table 3.2: **Summary of major file system workload studies over the past two decades.** For each study, we show the date of trace collection, the file system or protocol studied, whether it involved network file systems, and the workloads studied.

Early file system workload studies, such as those of the BSD [123], VAX/VMS [131], and Sprite [17] file systems, revealed a number of important observations and trends that guided file system design for the last two decades. In particular, they observed a significant tendency towards large, sequential read access, limited read-write access, bursty I/O patterns, and very short file lifetimes. Another study observed workload self-similarity during short time periods, though not for long time periods [71]. A more recent study of the Windows NT file system [177] supported a number of the past observations and trends. In 2000, Roselli, *et al.* compared four file system workloads [137]; they noted that block lifetimes had increased since past studies and explored the effect on caching strategies. At the time of our study, the most recent study

was from 2001 and analyzed NFS traffic to network file servers [48], identifying a number of peculiarities with NFS tracing and arguing that pathnames can aid file system layout. Since we conducted our study, another study that examined NFS traces from an intensive computer animation environment was published [10].

3.1.1 The Need for a New Study

Although there have been a number of previous file system workload studies, several factors indicate that a new workload study may aid ongoing network file system design.

Time since last study. There have been significant changes in computing power, network bandwidth, and network file system usage since the last major study in 2001 [48]. A new study will help understand how these changes impact network file system workloads.

Few network file system studies. Only a few studies have explored network file system workloads [17, 48, 131], despite their differences from local file systems. Local file systems workloads include the access patterns of many system files, which are generally read-only and sequential, and are focused on the client's point of view. While such studies are useful for understanding client workload, it is critical for network file systems to focus on the workload seen at the server, which often excludes system files or accesses that hit the client cache.

No CIFS protocol studies. The only major network file system study in the past decade analyzed NFSv2 and v3 workloads [48]. Though NFS is common on UNIX systems, most Windows systems use CIFS. Given the widespread Windows client population and differences between CIFS and NFS (*e. g.*, CIFS is a stateful protocol, in contrast to NFSv2 and v3), analysis beyond NFS can add more insight into network file system workloads.

Limited file system workloads. University [17, 48, 71, 123, 137] and personal computer [191] workloads have been the focus of a number of past studies. While useful, these workloads may differ from the workloads of network file systems deployed in other environments.

3.1.2 The CIFS Protocol

The CIFS protocol, which is based on the Server Message Block (SMB) protocol that defines most of the file transfer operations used by CIFS, differs in a number of respects from oft-studied NFSv2 and v3. Most importantly, CIFS is stateful: CIFS user and file operations act on stateful session IDs and file handles, making analysis of access patterns simpler and more accurate than in NFSv2 and v3, which require heuristics to infer the start and end of an access [48]. Although CIFS may differ from other protocols, we believe our observations are *not* tied exclusively to CIFS. Access patterns, file sharing, and other workload characteristics observed by the file server are influenced by the file system users, their applications, and the behavior of the file system client, which are not closely tied to the transfer protocol.

3.2 Workload Tracing Methodology

We collected CIFS network traces from two large-scale, enterprise-class file servers deployed in the NetApp corporate headquarters. One is a mid-range file server with 3 TB of total storage, with almost 3 TB used, deployed in the corporate data center that hosts data used by over 1000 marketing, sales, and finance employees. The other is a high-end file server with over 28 TB of total storage, with 19 TB used, deployed in the engineering data center. It is used by over 500 engineering employees. Throughout the rest of this paper, we refer to these workloads as *corporate* and *engineering*, respectively.

All NetApp storage servers support multiple protocols including CIFS, NFS, iSCSI, and Fibre Channel. We traced *only* CIFS on each file server. For the corporate server, CIFS was the primary protocol used, while the engineering server saw a mix of CIFS and NFS protocols. These servers were accessed by desktops and laptops running primarily Windows for the corporate environment and a mix of Windows and Linux for the engineering environment. A small number of clients also ran Mac OS X and FreeBSD. Both servers could be accessed through a Gigabit Ethernet LAN, a wireless LAN, or via a remote VPN.

The traces were collected from both the corporate and engineering file servers between August 10th and December 14th, 2007. For each server, we mirrored a port on the

network switch to which it was connected and attached it to a Linux workstation running `tcpdump` [169]. Since CIFS often utilizes NetBIOS [3], the workstation recorded all file server traffic on the NetBIOS name, datagram, and session service ports as well as traffic on the CIFS port. The trace data was periodically copied to a separate file server. Approximately 750 GB and 1.5 TB of uncompressed `tcpdump` traces were collected from the corporate and engineering servers, respectively. All traces were post-processed with `tshark 0.99.6` [185], a network packet analyzer. All filenames, usernames, and IP addressed were anonymized.

Our analysis of CIFS presented us with a number of challenges. CIFS is a stream-based protocol, so CIFS headers do not always align with TCP packet boundaries. Instead, CIFS relies on NetBIOS to define the length of the CIFS command and data. This became a problem during peak traffic periods when `tcpdump` dropped a few packets, occasionally causing a NetBIOS session header to be lost. Without the session header, `tshark` was unable to locate the beginning of the next CIFS packet within the TCP stream, though it was able to recover when it found a new session header aligned with the start of a TCP packet. To recover CIFS requests that fell within this region, we wrote a program to parse the `tcpdump` data and extract complete CIFS packets based on a signature of the CIFS header while ignoring any NetBIOS session information.

Another issue we encountered was the inability to correlate CIFS sessions to user-names. CIFS is a session based protocol in which a user begins a session by connecting to the file server via an authenticated login process. However, authentication in our trace environment almost always uses Kerberos [118]. Thus, regardless of the actual user, user authentication cre-dentials are cryptographically changed with each login. As a result, we were unable to match a particular user across multiple sessions. Instead we relied on the client's IP address to correlate users to sessions. While less accurate, it provides a reasonable estimate of users since most users access the servers via the LAN with the same IP address.

3.3 Workload Analysis

This section presents the results of our analysis of our corporate and engineering CIFS workloads. We first describe the terminology used throughout the analysis. Our analysis begins with a comparison of our workloads and then a comparison to past studies. We then analyze workload activity with a focus on I/O and sharing distributions. Finally, we examine properties of file type and user session access patterns. We italicize our key observations following the section in which they are discussed.

3.3.1 Terminology

Our study relies on several frequently used terms to describe our observations. Thus, we begin by defining the following terms:

I/O A single CIFS read or write command.

Sequential I/O An I/O that immediately follows the previous I/O to a file within an open/close pair (i.e., its offset equals the sum of the previous I/O's offset and length). The first I/O to an open file is always considered sequential.

Random I/O An I/O that is not sequential.

Sequential Run A series of sequential I/Os. An opened file may have multiple sequential runs.

Sequentiality Metric The fraction of bytes transferred sequentially. This metric was derived from a similar metric described by Ellard, *et al.* [48].

Open Request An open request for a file that has at least one subsequent I/O and for which a close request was observed. Some CIFS metadata operations cause files to be opened without ever being read or written. These open requests are artifacts of the CIFS client implementation, rather than the workload, and are thus excluded.

Client A unique IP address. Since Kerberos authentication prevents us from correlating usernames to users, we instead rely on IP address to identify unique clients.

3.3.2 Workload Comparison

Table 3.3 shows a summary comparison of overall characteristics for both corporate and engineering workloads. For each workload we provide some general statistics along with the frequency of each CIFS request. Table 3.3 shows that engineering has a greater number of requests, due to a longer tracing period, though, interestingly, both workloads have similar request percentages. For both, about 21% of requests are file I/O and about 50% are metadata operations. There are also a number of CIFS-specific requests. The I/O percentages differ from NFS workloads, in which 30–80% of all operations were I/O [48, 159]. This difference can likely be attributed to both differences in workload and protocol.

The total data transferred in the two traces combined was just over 1.6 TB of data, which is less than 10% of the file servers' active storage of over 22 TB of data. Since the data transfer summaries in Table 3.3 include files that were transferred multiple times, our observations show that somewhat more than 90% of the active storage on the file servers was untouched over the three month trace period.

The read/write byte ratios have decreased significantly compared to past studies [17, 48, 137]. We found only a 2:1 ratio indicating workloads are becoming less read-centric, in contrast to past studies that found ratios of 4:1 or higher. We believe that a key reason for the decrease in the read-write ratio is that client caches absorb a significant percentage of read requests. It is also interesting that the corporate and engineering request mix are similar, perhaps because of similar work being performed on the respective clients (*e. g.*, Windows office workloads) or because client caching and I/O scheduling obfuscate the application and end-user behavior. **Observation 1:** *Both of the workloads are more write-heavy than workloads studied previously.*

Figures 3.1(a) and 3.1(b) show the distribution of total CIFS requests and I/O requests for each workload over a one week period and a nine week period, respectively. Figure 3.1(a) groups request counts into hourly intervals and Figure 3.1(b) uses daily intervals. Figure 3.1(a) shows, unsurprisingly, that both workloads have strongly diurnal cycles and that there are very evident peak and idle periods throughout a day. The cyclic idle periods show there are op-

	Corporate	Engineering
Clients	5261	2654
Days	65	97
Data read (GB)	364.3	723.4
Data written (GB)	177.7	364.4
R:W I/O ratio	3.2	2.3
R:W byte ratio	2.1	2.0
Total operations	228 million	352 million
Operation name	%	%
Session create	0.4	0.3
Open	12.0	11.9
Close	4.6	5.8
Read	16.2	15.1
Write	5.1	6.5
Flush	0.1	0.04
Lock	1.2	0.6
Delete	0.03	0.006
File stat	36.7	42.5
Set attribute	1.8	1.2
Directory read	10.3	11.8
Rename	0.04	0.02
Pipe transactions	1.4	0.2

Table 3.3: **Summary of trace statistics.** File system operations broken down by workload. All operations map to a single CIFS command except for file stat (composed of `query_path_info` and `query_file_info`) and directory read (composed of `find_first2` and `find_next2`). Pipe transactions map to remote IPC operations.

portunities for background processes, such as log-cleaning and disk scrubbing to run without interfering with user requests.

Interestingly, there is a significant amount of variance between individual days in the number and ratio of both requests and I/O. In days where the number of total requests are increased, the number of read and write requests are not necessarily increased. This is also the case between weeks in Figure 3.1(b). The variation between total requests and I/O requests implies any that single day or week is likely an insufficient profile of the overall workload, so it is probably inaccurate to extrapolate trace observations from short time periods to longer periods, as was also noted in past studies [48, 71, 174, 177]. It is interesting to note that the overall request mix presented in Table 3.3 is different from the mix present in any single day or week, suggesting that the overall request mix might be different if a different time period were traced and is influenced more by workload than by behavior of the file system client.

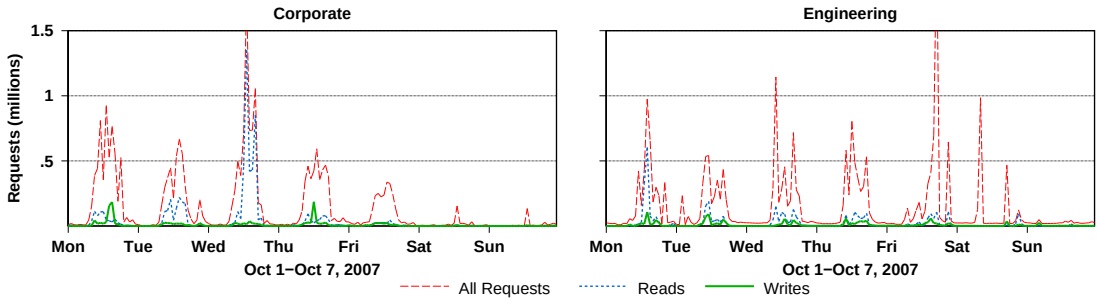
3.3.3 Comparison to Past Studies

We now compare our CIFS network file system workloads with those of past studies, including those in NFS [48], Sprite [17], VxFS [137] and Windows NT [177] studies. In particular, we analyze how file access patterns and file lifetimes have changed. For comparison purposes, we use tables and figures consistent with those of past studies.

3.3.3.1 File Access Patterns

Table 3.4 provides a summary comparison of file access patterns, showing access patterns in terms of both I/O requests and bytes transferred. Access patterns are categorized by whether a file was accessed read-only, write-only, or read-write. Sequential access is divided into two categories, *entire* accesses, which transfer the entire file, and *partial* accesses, which do not.

Table 3.4 shows a remarkable increase in the percentage of read-write I/O and bytes transferred. Most previous studies observed less than 7% of total bytes transferred to files accessed read-write. However, both our corporate and engineering workloads have over 20% of bytes transferred in read-write accesses. Furthermore, 45.9% of all corporate I/Os and 32.1%



(a) Requests over a week.

(b) Requests over 9 weeks.

Figure 3.1: **Request distribution over time.** The frequency of all requests, read requests, and write requests are plotted over time. Figure 3.1(a) shows how the request distribution changes for a single week in October 2007. Here request totals are grouped in one hour intervals. The peak one hour request total for corporate is 1.7 million and 2.1 million for engineering. Figure 3.1(b) shows the request distribution for a nine week period between September and November 2007. Here request totals are grouped into one day intervals. The peak one day intervals are 9.4 million for corporate and 19.1 million for engineering.

File System Type	Network							Local		
Workload	Corporate		Engineering		CAMPUS	EECS	Sprite	Ins	Res	NT
Access Pattern	I/Os	Bytes	I/Os	Bytes	Bytes	Bytes	Bytes	Bytes	Bytes	Bytes
Read-Only (% total)	39.0	52.1	50.6	55.3	53.1	16.6	83.5	98.7	91.0	59.0
Entire file sequential	13.5	10.5	35.2	27.4	47.7	53.9	72.5	86.3	53.0	68.0
Partial sequential	58.4	69.2	45.0	55.0	29.3	36.8	25.4	5.9	23.2	20.0
Random	28.1	20.3	19.8	17.6	23.0	9.3	2.1	7.8	23.8	12.0
Write-Only (% total)	15.1	25.2	17.3	23.6	43.8	82.3	15.4	1.1	2.9	26.0
Entire file sequential	21.2	36.2	15.6	35.2	37.2	19.6	67.0	84.7	81.0	78.0
Partial sequential	57.6	55.1	63.4	61.0	52.3	76.2	28.9	9.3	16.5	7.0
Random	21.2	8.7	21.0	3.8	10.5	4.1	4.0	6.0	2.5	15.0
Read-Write (% total)	45.9	22.7	32.1	21.1	3.1	1.1	1.1	0.2	6.1	15.0
Entire file sequential	7.4	0.1	0.4	0.1	1.4	4.4	0.1	0.1	0.0	22.0
Partial sequential	48.1	78.3	27.5	50.0	0.9	1.8	0.0	0.2	0.3	3.0
Random	44.5	21.6	72.1	49.9	97.8	93.9	99.9	99.6	99.7	74.0

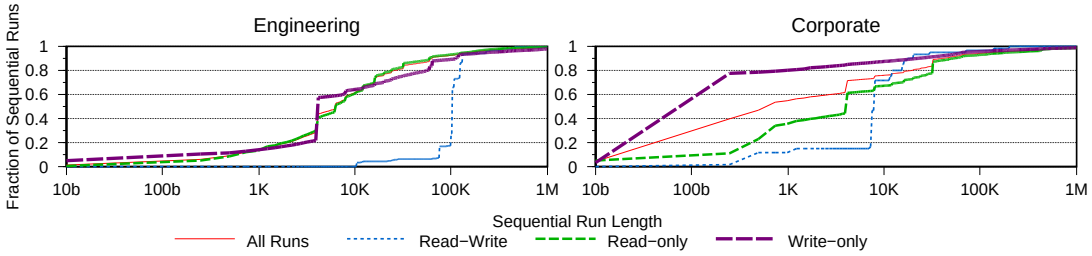
Table 3.4: **Comparison of file access patterns.** File access patterns for the corporate and engineering workloads are compared with those of previous studies. CAMPUS and EECS [48] are university NFS mail server and home directory workloads, respectively. Both were measured in 2001. Sprite [17], Ins and Res [137] are university computer lab workloads. Sprite was measured in 1991 and Ins and Res were measured between 1997 and 2000. NT [177] is a combination of development and scientific workloads measured in 1998.

of all engineering I/Os are in read-write accesses. This shows a diversion from the read-only oriented access patterns of past workloads. When looking closer at read-write access patterns we find that sequentiality has also changed; 78.3% and 50.0% of bytes are transferred sequentially as compared to roughly 1% of bytes in past studies. However, read-write patterns are still very random relative to read-only and write-only patterns. These changes may suggest that network file systems store a higher fraction of mutable data, such as actively changing documents, which make use of the centralized and shared environment and a smaller fraction of system files, which tend to have more sequential read accesses. These changes may also suggest that the sequential read-oriented patterns which some file systems are designed [105] for are less prevalent in network file systems, and write-optimized file systems [77, 138] may be better suited. **Observation 2:** *Read-write access patterns are much more frequent compared to past studies.*

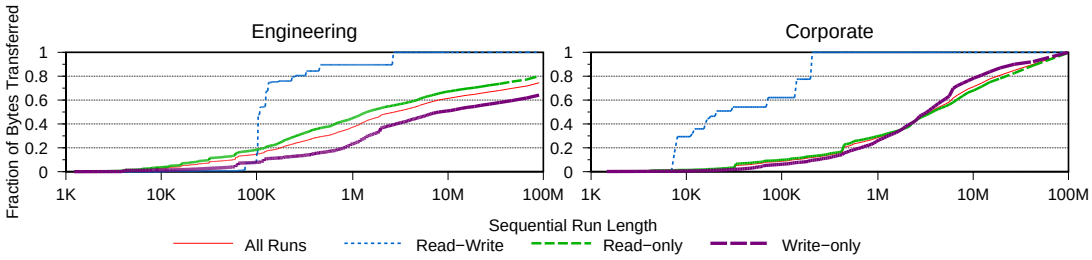
Another interesting observation is that few fewer files have bytes transferred from the entire file, that is, the whole file is read or written from beginning to end. In past studies, well over 50% of read-only and write-only files are accessed in their entirety. However, we found only 10 and 27.4% of read-only bytes in corporate and engineering for which this is the case and only 36.2 and 35.2% of write-only files.

3.3.3.2 Sequentiality Analysis

Next, we compared the sequential access patterns found in our workloads with past studies. A sequential run is defined as a series of sequential I/Os to a file. Figure 3.2(a) shows the distribution of sequential run lengths. We see that sequential runs are short for both workloads, with almost all runs shorter than 100 KB, which is consistent with past studies. This observation suggests that file systems should continue to optimize for short sequential common-case accesses. However, Figure 3.2(b), which shows the distribution of bytes transferred during sequential runs, has a very different implication, indicating that many bytes are transferred in long sequential runs: between 50–80% of bytes are transferred in runs of 10 MB or less. In addition, the distribution of sequential runs for the engineering workload is long-tailed, with 8% of bytes transferred in runs longer than 400 MB. Interestingly, read-write sequential runs exhibit



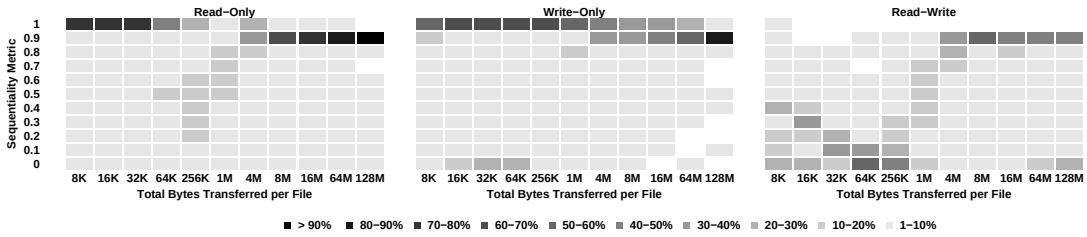
(a) Length of sequential runs.



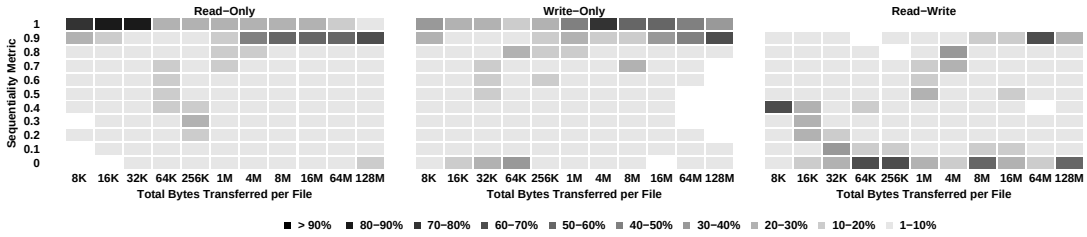
(b) Bytes transferred in sequential runs.

Figure 3.2: **Sequential run properties.** Sequential access patterns are analyzed for various sequential run lengths. The X-axes are given in a log-scale. Figure 3.2(a) shows the length of sequential runs, while Figure 3.2(b) shows how many bytes are transferred in sequential runs.

very different characteristics from read-only and write-only runs: Most read-write bytes are transferred in much smaller runs. This implies that the interactive nature of read-write accesses is less prone to very large transfers, which tend to be mostly read-only or write-only. Overall, we found that most bytes are transferred in much larger runs—up to 1000 times longer—when compared to those observed in past studies, though most runs are short. Our results suggest file systems must continue to optimize for small sequential access, though they must be prepared to handle a small number of very large sequential accesses. This also correlates with the heavy-tailed distributed of file sizes, which is discussed later; for every large sequential run there must be at least one large file. **Observation 3:** *Bytes are transferred in much longer sequential runs than in previous studies.*



(a) Corporate.

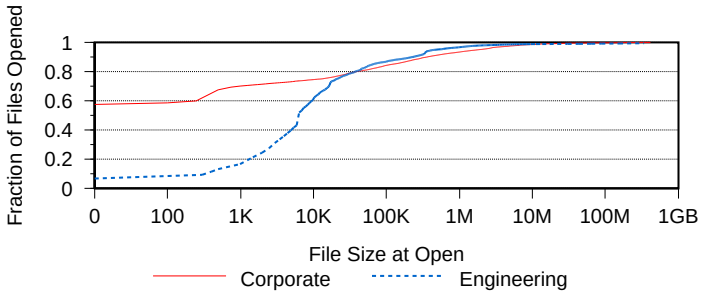


(b) Engineering.

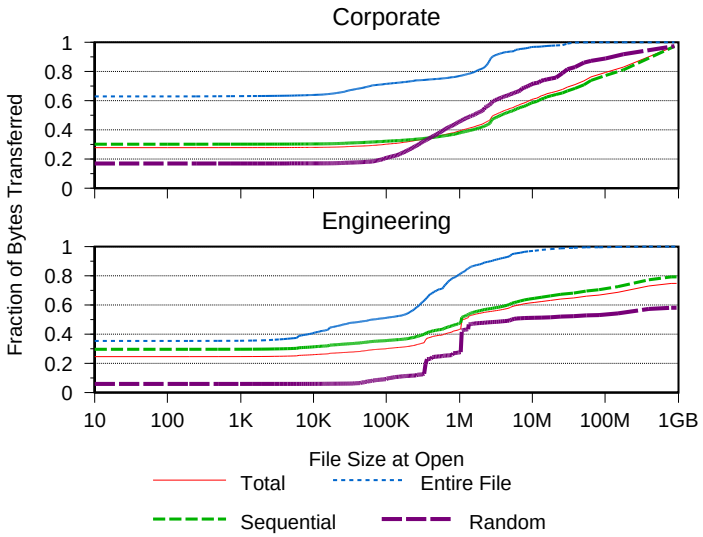
Figure 3.3: **Sequentiality of data transfer**. The frequency of sequentiality metrics is plotted against different data transfer sizes. Darker regions indicate a higher fraction of total transfers. Lighter regions indicate a lower fraction. Transfer types are broken into read-only, write-only, and read-write transfers. Sequentiality metrics are grouped by tenths for clarity.

We now examine the relationship between request sizes and sequentiality. In Figure 3.3(a) and Figure 3.3(b) we plot the number of bytes transferred from a file against the sequentiality of the transfer. This is measured using the sequentiality metric: The fraction of bytes transferred sequentially, with values closer to one meaning higher sequentiality. Figures 3.3(a) and 3.3(b) show this information in a heat map in which darker regions indicate a higher fraction of transfers with that sequentiality metric and lighter regions indicate a lower fraction. Each region within the heat map represents a 10% range of the sequentiality metric. We see from Figures 3.3(a) and 3.3(b) that small transfers and large transfers are more sequential for read-only and write-only access, which is the case for both workloads. However, medium-sized transfers, between 64 KB and 4 MB, are more random. For large and small transfers, file systems may be able to anticipate high sequentiality for read-only and write-only access. Read-write accesses, on the other hand, are much more random for most transfer sizes. Even very large read-write transfers are not always very sequential, which follows from our previous observations in Figure 3.2(b), suggesting that file systems may have difficulty anticipating the sequentiality of read-write accesses.

Next, we analyze the relationship between file size and access pattern by examining the size of files at open time to determine the most frequently opened file sizes and the file sizes from which most bytes are transferred. It should be noted that since we only look at opened files, it is possible that this does not correlate to the file size distribution across the file system. Our results are shown in Figures 3.4(a) and 3.4(b). In Figure 3.4(a) we see that 57.5% of opens in the corporate workload are to newly-created files or truncated files with zero size. However, this is not the case in the engineering workload, where only 6.3% of opens are to zero-size files. Interestingly, both workloads find that most opened files are small; 75% of opened files are smaller than 20 KB. However, Figure 3.4(a) shows that most bytes are transferred from much larger files. In both workloads we see that only about 60% of bytes are transferred from files smaller than 10 MB. The engineering distribution is also long-tailed with 12% of bytes being transferred from files larger than 5 GB. By comparison, almost all of the bytes transferred in previous studies came from files smaller than 10 MB. These observations suggest that larger files play a more significant role in network file system workloads than in those previously



(a) Open requests by file size.



(b) Bytes transferred by file size.

Figure 3.4: **File size access patterns**. The distribution of open requests and bytes transferred are analyzed according to file size at open. The X-axes are shown on a log-scale. Figure 3.4(a) shows the size of files most frequently opened. Figure 3.4(b) shows the size of files from which most bytes are transferred.

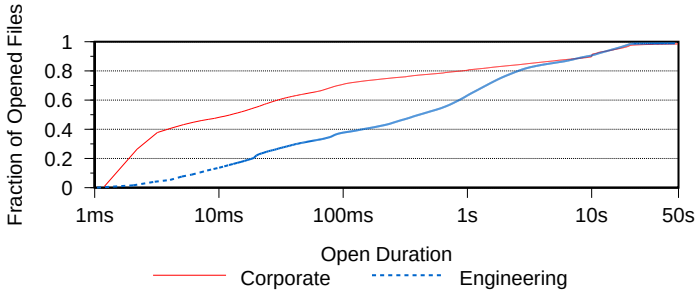


Figure 3.5: **File open durations.** The duration of file opens is analyzed. The X-axis is presented on a log-scale. Most files are opened very briefly, although engineering files are opened slightly longer than corporate files.

studied. This may be due to frequent small file requests hitting the local client cache. Thus, file systems should still optimize small file layout for frequent access and large file layout for large transfers. **Observation 4:** *Bytes are transferred from much larger files than in previous studies.*

Figure 3.5 shows the distribution of file open durations. We find that files are opened for shorter durations in the corporate workload than in the engineering workload. In the corporate workload, 71.1% of opens are shorter than 100 ms, but just 37.1% are similarly short in the engineering workload. However, for both workloads most open durations are less than 10 seconds, which is similar to observations in past studies. This is also consistent with our previous observations that small files, which likely have short open durations, are most frequently accessed.

3.3.3.3 File Lifetime

This section examines how file lifetimes have changed compared to past studies. In CIFS, files can be either deleted through an explicit delete request, which frees the entire file and its name, or through truncation, which only frees the data. Figure 3.6 shows the distribution of file lifetimes, broken down by deletion method. We find that most created files live longer than 24 hours, with 57.0% and 64.9% of corporate and engineering files persisting for more than a day. Both distributions are long-tailed, meaning many files live well beyond 24 hours. However, files that *are* deleted usually live less than a day: only 18.7% and 6.9% of eventually deleted

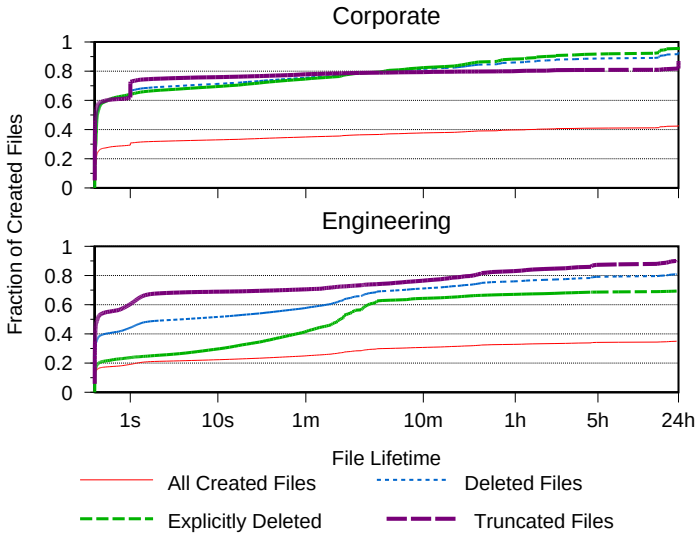
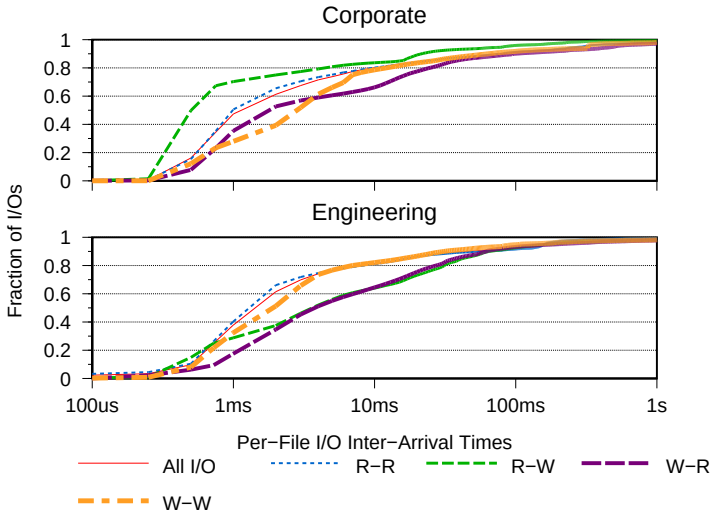


Figure 3.6: **File lifetimes.** The distributions of lifetimes for all created and deleted files are shown. Time is shown on the X-axis on a log-scale. Files may be deleted through explicit delete request or truncation.

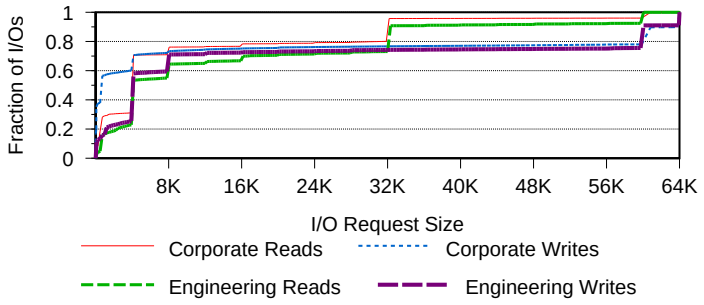
files live more than 24 hours. Nonetheless, compared to past studies in which almost all deleted files live less than a minute, deleted files in our workloads tend to live much longer. This may be due to fewer temporary files being created over the network. However, we still find that some files live very short lifetimes. In each workload, 56.4% and 38.6% of deleted files are deleted within 100 ms of creation, indicating that file systems should expect fewer files to be deleted and files that live beyond a few hundred milliseconds to have long lifetimes. **Observation 5** *Files live an order of magnitude longer than in previous studies.*

3.3.4 File I/O Properties

We now take a closer look at the properties of file I/O where, as defined in Section 3.3.1, an I/O request is defined as any single read or write operation. We begin by looking at per-file, per-session I/O inter-arrival times, which include network round-trip latency. Intervals are categorized by the type of requests (read or write) that bracket the interval; the distribution of interval lengths is shown in Figure 3.7(a). We find that most inter-arrival times are between



(a) Per-file I/O inter-arrival times.



(b) I/O request sizes.

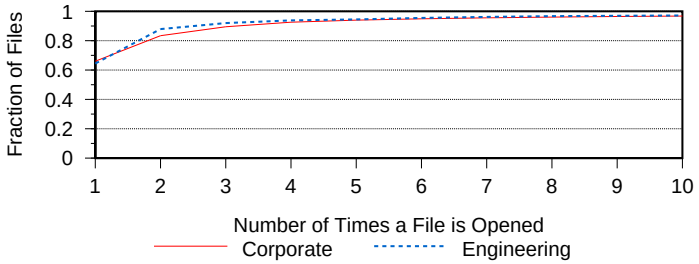
Figure 3.7: **File I/O properties**. The burstiness and size properties of I/O requests are shown. Figure 3.7(a) shows the I/O inter-arrival times. The X-axis is presented on a log-scale. Figure 3.7(b) shows the sizes of read and write I/O. The X-axis is divided into 8 KB increments.

100 μs and 100 ms. In fact, 96.4% and 97.7% of all I/Os have arrival times longer than 100μs and 91.6% and 92.4% are less than 100 ms for corporate and engineering, respectively. This tight window means that file systems may be able to make informed decisions about when to prefetch or flush cache data. Interestingly, there is little distinction between read-read or read-write and write-read or write-write inter-arrival times. Also, 67.5% and 69.9% of I/O requests have an inter-arrival time of less than 3 ms, which is shorter than some measured disk response times [134]. These observations may indicate cache hits at the server or possibly asynchronous I/O. It is also interesting that both workloads have similar inter-arrival time distributions even though the hardware they use is of different classes, a mid-range model versus a high-end model.

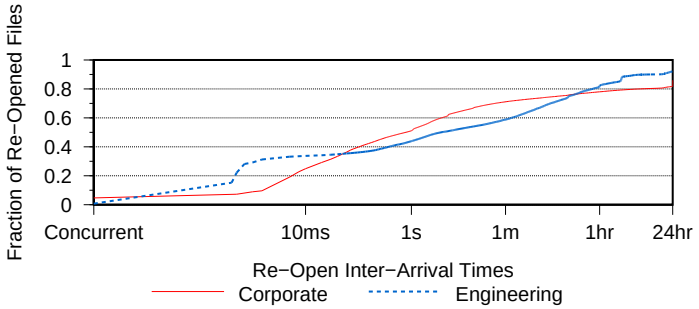
Next, we examine the distribution of bytes transferred by a single I/O request. As Figure 3.7(b) shows, most requests transfer less than 8 KB, despite a 64 KB maximum request size in CIFS. This distribution may vary between CIFS and NFS since each buffers and schedules I/O differently. The distribution in Figure 3.7(b) increases for only a few I/O sizes, indicating that clients generally use a few specific request sizes. This I/O size information can be combined with the I/O inter-arrival times from Figure 3.7(a) to calculate a distribution of I/Os per-second (IOPS) that may help file systems determine how much buffer space is required to support various I/O rates.

3.3.5 File Re-Opens

In this section, we explore how frequently files are re-opened, *i. e.*, opened more than once during the trace period. Figure 3.8(a), shows the distribution of the number of times a file is opened. For both workloads, we find that the majority of files, 65%, are opened only once during the entire trace period. The infrequent re-access of many files suggests there are opportunities for files to be archived or moved to lower-tier storage. Further, we find that about 94% of files are accessed fewer than five times. However, both of these distributions are long-tailed—some files are opened well over 100,000 times. These frequently re-opened files account for about 7% of total opens in both workloads. **Observation 6:** *Most files are not re-opened once they are closed.*



(a) File open frequencies.



(b) Re-open inter-arrival times.

Figure 3.8: **File open properties**. The frequency of and duration between file re-opens is shown. Figure 3.8(a) shows how often files are opened more than once. Figure 3.8(b) shows the time between re-opens and time intervals on the X-axis are given in a log-scale.

We now look at inter-arrival times between re-opens of a file. Re-open inter-arrival time is defined as the duration between the last close of a file and the time it is re-opened. A re-open is considered *concurrent* if a re-open occurs while the file is still open (*i. e.*,, it has not yet been closed). The distribution of re-open inter-arrival times is shown in Figure 3.8(b). We see that few re-opens are concurrent, with only 4.7% of corporate re-opens and 0.7% of engineering re-opens occurring on a currently-open file. However, re-opens *are* temporally related to the previous close; 71.1% and 58.8% of re-opens occur less than one minute after the file is closed. Using this information, file systems may be able to decide when a file should be removed from the buffer cache or when it should be scheduled for migration to another storage tier. **Observation 7:** *If a file is re-opened, it is temporally related to the previous close.*

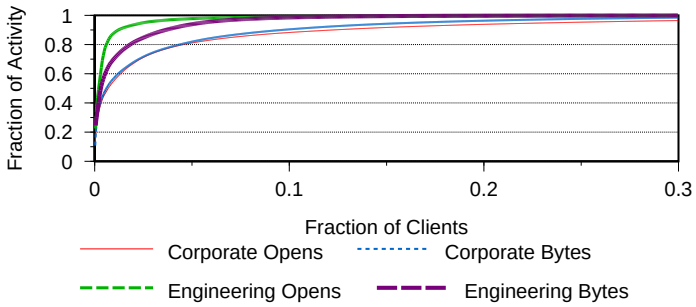


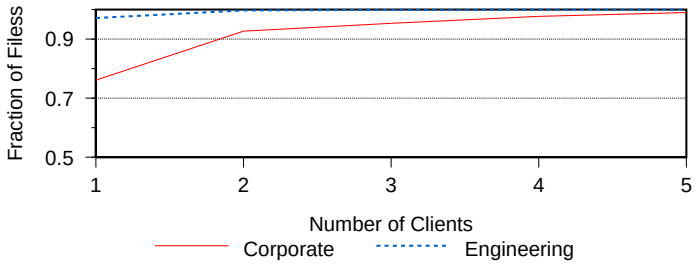
Figure 3.9: **Client activity distribution** The fraction of clients responsible for certain activities is plotted.

3.3.6 Client Request Distribution

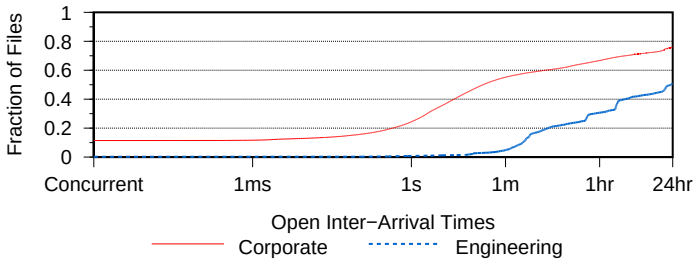
We next examine the distribution of file open and data requests amongst clients; recall from Section 3.3.1 that "client" refers to a unique IP address rather than an individual user. We use Lorenz curves [100]—cumulative distribution functions of probability distributions—rather than random variables to show the distribution of requests across clients. Our results, shown in Figure 3.9, find that a tiny fraction of clients are responsible for a significant fraction of open requests and bytes transferred. In corporate and engineering, 0.02% and 0.04% of clients make 11.9% and 22.5% of open requests and account for 10.8% and 24.6% of bytes transferred, respectively. Interestingly, 0.02% of corporate clients and 0.04% of engineering clients correspond to approximately 1 client for each workload. Additionally, we find that about 35 corporate clients and 5 engineering clients account for close to 50% of the opens in each workload. This suggests that the distribution of activity is highly skewed and that file systems may be able to take advantage of this information by doing informed allocation of resources or quality of service planning. **Observation 8:** *A small fraction of clients account for a large fraction of file activity.*

3.3.7 File Sharing

This section looks at the extent of file sharing in our workloads. A file is shared when two or more clients open the same file *some time* during the trace period; the sharing need not



(a) File sharing frequencies.



(b) Sharing inter-arrival times.

Figure 3.10: **File sharing properties**. We analyze the frequency and temporal properties of file sharing. Figure 3.10(a) shows the distribution of files opened by multiple clients. Figure 3.10(b) shows the duration between shared opens. The durations on the X-axis are in a log-scale.

be concurrent. Since we can only distinguish IP addresses and not actual users, it is possible that two IP addresses may represent a single (human) user and vice-versa. However, the drastic skew of our results indicates this likely has little impact on our observations. Also, we only consider opened files; files which have only had their metadata accessed by multiple clients are not included in the these results.

Figure 3.10(a) shows the distribution of the frequency with which files are opened by multiple clients. We find that most files are only opened by a single client. In fact, 76.1% and 97.1% of files are only opened by one client in corporate and engineering, respectively. Also, 92.7% and 99.7% of files are ever opened by two or fewer clients. This suggests that the shared environment offered by network file systems is not often taken advantage of. Other methods of sharing files, such as email, web and Wiki pages, and content repositories (*e. g.*, `svn` and `git`),

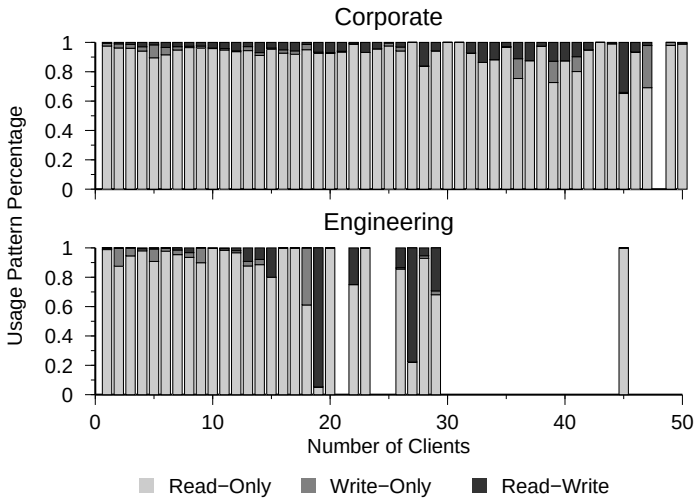


Figure 3.11: **File sharing access patterns.** The fraction of read-only, write-only, and read-write accesses are shown for differing numbers of sharing clients. Gaps are seen where no files were shared with that number of clients.

may reduce the need for clients to share files via the file system. However, both distributions are long-tailed, and a few files are opened by many clients. In the corporate workload, four files are opened by over 2,000 clients and in the engineering workload, one file is opened by over 1,500 clients. This shows that, while not common, sharing files through the file system can be heavily used on occasion. **Observation 9:** *Files are infrequently accessed by more than one client.*

In Figure 3.10(b) we examine inter-arrival times between different clients opening a file. We find that concurrent (simultaneous) file sharing is rare. Only 11.4% and 0.2% of shared opens from different clients were concurrent in corporate and engineering, respectively. When combined with the observation that most files are only opened by a single client, this suggests that synchronization for shared file access is not often required, indicating that file systems may benefit from looser locking semantics. However, when examining the duration between shared opens we find that sharing does have a temporal relationship in the corporate workload; 55.2% of shared opens occur within one minute of each other. However, this is not true for engineering, where only 4.9% of shared opens occur within one minute.

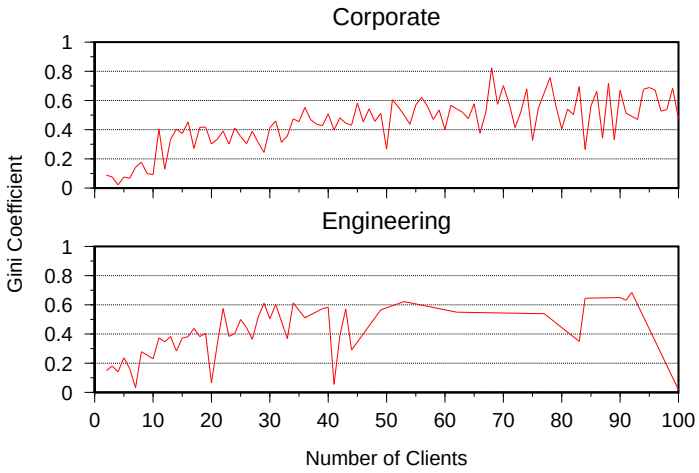


Figure 3.12: **User file sharing equality**. The equality of sharing is shown for differing numbers of sharing clients. The Gini coefficient, which measures the level of equality, is near 0 when sharing clients have about the same number of opens to a file. It is near 1 when clients unevenly share opens to a file.

We now look at the manner (read-only, write-only, or read-write) with which shared files are accessed. Figure 3.11 shows the usage patterns for files opened by multiple clients. Gaps are present where no files were opened by that number of clients. We see that shared files are accessed read-only the majority of the time. These may be instances of reference documents or web pages that are rarely re-written. The number of read-only accesses slightly decreases as more clients access a file and a read-write pattern begins to emerge. This suggests that files accessed by many clients are more mutable. These may be business documents, source code, or web pages. Since synchronization is often only required for multiple concurrent writers, these results further argue for loose file system synchronization mechanisms. **Observation 10:** *File sharing is rarely concurrent and mostly read-only.*

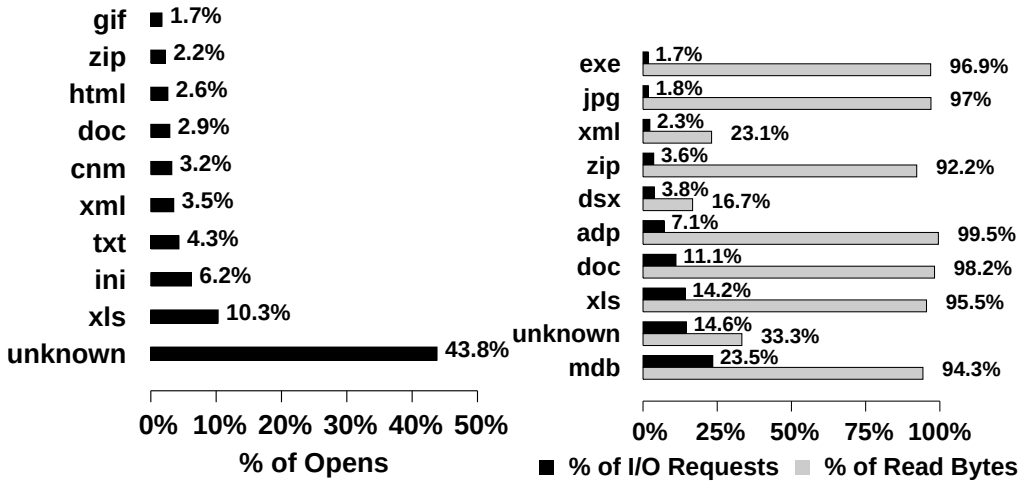
Finally, we analyze which clients account for the most opens to shared files. Equality measures how open requests are distributed amongst clients sharing a file. Equal file sharing implies all sharing clients open the shared file an equal number of times. To analyze equality, we use the Gini coefficient [64], which measures statistical dispersion, such as the inequality

of income in economic analysis. We apply the equality concept to how frequently a shared file is opened by a client. Lower coefficients mean sharing clients open the file more equally (the same number of times), and higher coefficients mean a few clients account for the majority of opens. Figure 3.12 shows Gini coefficients for various numbers of shared clients. We see that as more clients open a file, the level of equality decreases, meaning that fewer clients begin to dominate the number of open requests. Gini coefficients are lower, less than 0.4, for files opened by fewer than 20 clients, meaning that when a few clients access a file, they each open the file an almost equal number of times. As more clients access the file, a small number of clients begin to account for most of the opens. This may indicate that as more clients share a file, it becomes less reasonable for all sharing clients to access the file evenly, and a few dominant clients begin to emerge.

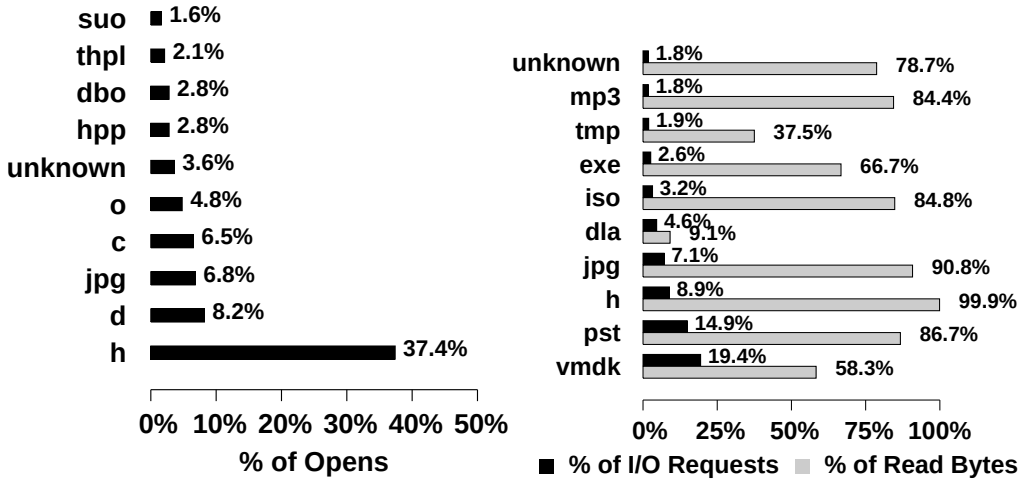
3.3.8 File Type and User Session Patterns

There have been a number of attempts to make layout, caching, and prefetching decisions based on how specific file types are accessed and the access patterns of certain users [50, 107]. In this section, we take a closer look at how certain file types are accessed and the access patterns that occur between when a user begins a CIFS "user session" by logging on and when they log-off. Our emphasis is on whether file types or users have common access patterns that can be exploited by the file system. We begin by breaking down file type frequencies for both workloads. Figures 3.13(a) and 3.13(b) show the most frequently opened and most frequently read and written file types. For frequently read and written file types, we show the fraction of bytes read for that type. Files with no discernible file extension are labeled as "unknown".

We find that the corporate workload has no file type, other than unknown types, that dominates open requests. However, 37.4% of all opens in the engineering workload are for C header files. Both workloads have a single file type that consumes close to 20% of all read and write I/O. Not surprisingly, these types correspond to generally large files, *e. g.*, `mdb` (Microsoft Access Database) files and `vmdk` (VMWare Virtual Disk) files. However, we find that most file types do not consume a significantly large fraction of open or I/O requests. This shows that file



(a) Corporate.



(b) Engineering.

Figure 3.13: **File type popularity.** The histograms on the right show which file types are opened most frequently. Those on the left show the file types most frequently read or written and the percentage of accessed bytes read for those types.

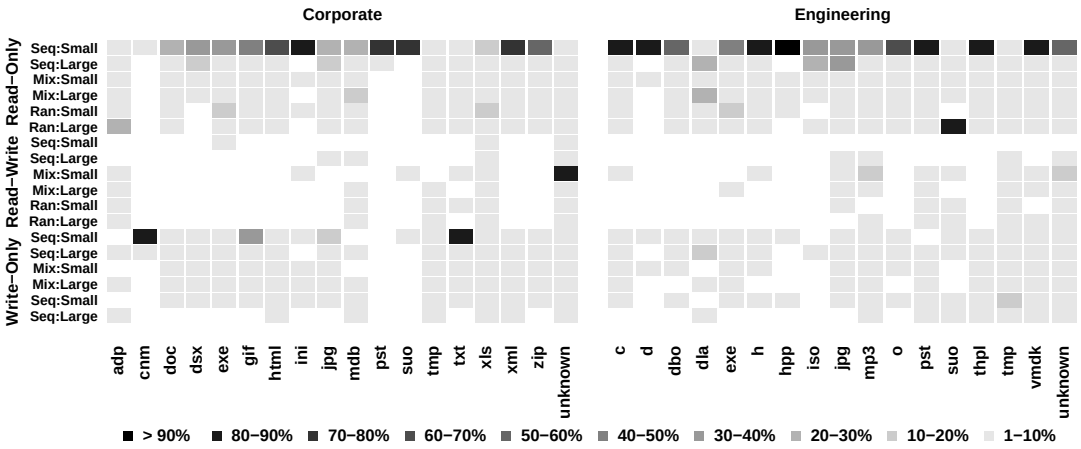


Figure 3.14: **File type access patterns** The frequency of access patterns are plotted for various file types. Access patterns are categorized into 18 groups. Increasingly dark regions indicate higher fractions of accesses with that pattern.

systems likely can be optimized for the small subset of frequently accessed file types. Interestingly, there appears to be little correlation between how frequently a file is opened and how frequently it is read or written. Only three corporate and two engineering file types appear as both frequently opened and frequently read or written; the `mdb` and `vmdk` types only constitute 0.5% and 0.08% of opens. Also, it appears file types that are frequently read or written are mostly read.

We now analyze the hypothesis that file systems can use file type and user access patterns to improve layout and prefetching [48, 50, 107]. We do so by examining *access signatures*, a vector containing the number of bytes read, bytes written, and sequentiality metric of a file access. We start by defining an access signature for each open/close pair for each file type above, we then apply K-means clustering [103] to the access signatures of each file type. K-means groups access signatures with similar patterns into unique clusters with varying densities. Our results are shown in Figure 3.14. For clarity we have categorized access signatures by the access type: read-only, write-only, or read-write. We further group signatures by their sequentiality metric ranges: 1–0.81 is considered highly sequential, 0.8–0.2 is considered mixed sequentiality, and 0.19–0 is considered highly random. Finally, access signatures are

categorized by the number of bytes transferred; access signatures are considered small if they transfer no more than 100 KB and large otherwise. Darker regions indicate the file type has a higher fraction of access signatures with those properties shown on the y-axis, and lighter regions indicate fewer signatures with those characteristics.

Figure 3.14 shows that most file types have several distinct kinds of access patterns, rather than one as previously presumed. Also, each type has multiple patterns that are more frequent than others, suggesting that file systems may not be able to properly predict file type access patterns using only a single pattern. Interestingly, small sequential read patterns occur frequently across most of the file types, implying that file systems should be optimized for this pattern, as is often already done. **Observation 11:** *Most file types do not have a single pattern of access.*

Surprisingly, file types such as `vmdk` that consume a large fraction of total I/Os are frequently accessed with small sequential reads. In fact, 91% of all `vmdk` accesses are this pattern, contradicting the intuition derived from Figure 3.13(b) that `vmdk` files have large accesses. However, a much smaller fraction of `vmdk` accesses transfer huge numbers of bytes in highly random read-write patterns. Several patterns read and write over 10 GB of data with a sequentiality metric less than 0.5, showing that frequent patterns may not be representative of the significant patterns in terms of bytes transferred or sequentiality. This argues that file systems should anticipate several patterns of access for any file type if layout or prefetching benefits are to be gained. Also, it is critical that they identify transitions between patterns. For example, a file system may, by default, prefetch data for `vmdk` files in small chunks: 100 KB or less. However, when over 100 KB of a `vmdk` file is accessed this signals the likely start of a very large transfer. In this case, the file system must properly adjust its prefetching.

Our observation that many file types exhibit several access patterns of varying frequency and significance draws an interesting comparison to the results in Table 3.4. Table 3.4 shows significant read-write I/O and byte transfer activity. However, file types in Figure 3.14 rarely have read-write patterns. This implies that read-write file accesses are in general uncommon however, when they do occur, a large number of bytes are accessed.

Next, we apply the same K-means clustering approach to access signatures of access patterns that occur within a CIFS user session. Recall that CIFS users begin a connection to the file server by creating an authenticated user session and end by eventually logging off. We define signatures for all accesses performed while the user is logged on. However, we only consider sessions in which bytes are transferred. The CIFS client opens short, temporary sessions for various auxiliary functions, which we exclude from this study as they do not represent a normal user log-in. Like file types, user sessions have several common patterns and no single pattern can summarize all of a user's accesses. The majority of user sessions have read-write patterns with less than 30 MB read and 10 MB written with a sequentiality metric close to 0.5, while a few patterns have much more significant data transfers that read and write gigabytes of data.

3.4 Design Implications

We now explore some of the possible implications of our workload analysis on network file system designs. We found that read-write access patterns have significantly increased relative to previous studies (see Section 3.3.3.1). Though we observed higher sequentiality in read-write patterns than past studies, they are still highly random compared to read-only or write-only access patterns (see Section 3.3.3.1). In contrast, a number of past studies found that most I/Os and bytes are transferred in read-only sequential access patterns [17, 123, 177], which has impacted the designs of several file systems [105, 116]. The observed shift towards read-write access patterns suggests file systems should look towards improving random access performance, perhaps through alternative media, such as flash. In addition, we observed that the ratio of data read to data written is decreasing compared to past studies [17, 48, 137] (see Section 3.3.2). This decrease is likely due to increasing effectiveness of client caches and fewer read-heavy system files on network storage. When coupled with increasing read-write access patterns, write-optimized file systems, such as LFS [138] and WAFL [77], or NVRAM write caching appear to be good designs for network file systems.

We observed that files are infrequently re-opened (see Section 3.3.5) and are usually accessed by only one client (see Section 3.3.7). This suggests that caching strategies which

exploit this, such as exclusive caching [186], may have practical benefits. Also, the limited reuse of files indicates that increasing the size of server data caches may add only marginal benefits. Rather, file servers may find larger metadata caches more valuable because metadata requests made up roughly 50% of all operations in both workloads, as Section 3.3.2 details.

The finding that most created files are not deleted (see Section 3.3.3.3) and few files are accessed more than once (see Section 3.3.5) suggests that many files may be good candidates for migration to lower-tier storage or archives. This is further motivated by our observation that only 1.6 TB were transferred from 22 TB of in-use storage over three months. While access to file metadata should be fast, this indicates much file data can be compressed, de-duplicated, or placed on low power storage, improving utilization and power consumption, without significantly impacting performance. In addition, our observation that file re-accesses are temporally correlated (see Section 3.3.5) means there are opportunities for intelligent migration scheduling decisions.

3.5 Previous Snapshot Studies

Like workload trace studies, snapshot studies have greatly influenced file system design [2, 57, 144]. Unlike workloads, snapshot studies analyze the properties of files at rest that make up the contents of the file system. In most cases, file metadata (*e. g.*, inode fields and extended attributes) attributes are studied. These studies are called snapshots because they represent a snapshot of the contents of the file system at a given point in time.

Table 3.5 summarizes the previous snapshot studies. Early file system snapshot studies focused on two key areas: file size and file lifetime. These two aspects were particularly important in early file system design because block allocation and space management algorithms were still in nascent stages. Two early studies on a local DEC machine [145] and NFS file servers [22] found that most files were less than several KB. For example, Satyanarayanan [145] found that 50% of files had fewer than 5 blocks (about 20 KB with a 4 KB block size), 95% had fewer than 100 blocks (about 400 KB), and 99% had less than 1000 blocks (about 4 MB). This distributions show a high skew towards small files. Additionally, functional lifetime, which is

Study	Date of Traces	FS/Protocol	Network FS	Environment
Satyanarayanan [145]	1981	DEC PDP-10		Academic lab
Bennett, *et al.* [22]	1991	NFS	x	Academic lab
Sienknecht, *et al.* [150]	1991-92	BSD		Corporate
Smith and Seltzer [153]	1994	FFS	x	Academic lab
Douceur and Bolosky [43]	1998	FAT, FAT32, NTFS		Engineering/PC
Agrawal, *et al.* [5]	2000-2004	FAT, FAT32, NTFS		Engineering/PC
Dayal [39]	2006	Various	x	HPC
Leung, *et al.*	2007	WAFL	x	Web, Engineering, Home

Table 3.5: **Summary of major file system snapshot studies over the past two decades.** For each study, the date of trace collection, the file system or protocol studied, whether it involved network file systems, and the kinds of workloads it hosted are shown.

the time between a file's last modification time and last access time, is generally short. Satyanarayanan found that 32% of files had a functional lifetime less than a day. However, Bennett, *et al.* [22] found that functional lifetimes were longer on their NFS file servers. A study of BSD file systems in a corporate setting [150] had similar findings which supported these observations. Another study [153] looked at snapshots to find how effectively FFS [105] allocates and organizes data on disk. Interestingly, they found that small files tend to more fragmented that larger files: Fewer than 35% of files with 2 blocks were optimally allocated though 85% of blocks in files larger than 64 blocks were allocated optimally.

More recently, two studies [5, 43] have examined five and ten thousand Windows personal computer file systems, respectively. These studies were significantly larger than previous studies and were the first to look at many snapshots from very similar environments, Windows desktop machines at Microsoft; 85% of theses systems used NTFS and the others used FAT and FAT32 systems. These studies have confirmed that many observations from older studies still hold true. For example, most files were still very small, less than 189 KB. However, they also observed changing trends since the previous studies, such as, the mean file size had increased from 108 KB to 189 KB. This finding corresponds with their observation that median file system capacities had increased from 5 GB to 40 GB. They also noted that directory size distributions

Data Set	Description	# of Files	Capacity
Web	web & Wiki server	15 million	1.28 TB
Eng	build space	60 million	30 GB
Home	home directories	300 million	76.78 TB

Table 3.6: **Metadata traces collected.** The small server capacity of the Eng trace is due to the majority of the files being small source code files: 99% of files are less than 1 KB.

Attribute	Description	Attribute	Description
inumber	inode number	owner	file owner
path	full path name	size	file size
ext	file extension	ctime	change time
type	file or directory	atime	access time
mtime	modification time	hlink	hard link #

Table 3.7: **Attributes used.** We analyzed the fields in the inode structure and extracted ext values from path.

had changed very little. They found that 90% directories still had two or fewer sub-directories and 20 or fewer entries.

3.6 Snapshot Tracing Methodology

Our file metadata snapshot traces were collected from three large-scale, enterprise-class file servers in the NetApp corporate headquarters. One hosts web and Wiki server files, another is a engineering build server, and another stores employee home directories. The size of these traces, which we refer to as Web, Eng, and Home, respectively, are described in Table 3.6. They represent over a quarter of a billions files and over 80 TB of actively used storage. The traces were collected using a program we wrote that performs a parallelized crawl of the namespace and collects metadata using the `stat()` system call. The crawls were performed

during the summer of 2007. The attributes that we collected are show in Table 3.7. NetApp servers support extended attributes, though they were rarely used in these traces and were thus ignored.

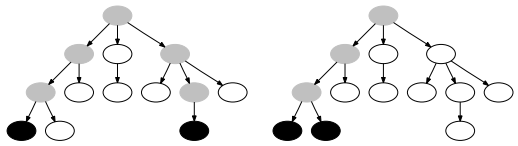
3.7 Snapshot Analysis

Our analysis revealed two key properties that we focus on: Metadata has *spatial locality* and highly *skewed distributions* of values.

3.7.1 Spatial Locality

Spatial locality means that metadata attribute values are clustered in the namespace (*i. e.*, occurring in relatively few directories). For example, `Andrew`'s files reside mostly in directories in the `/home/andrew` sub-tree, not scattered evenly across the namespace. Thus, files with `owner` equal to `andrew` likely occur in only a small fraction of the total directories in the file system. Spatial locality comes from the way that users and applications organize files in the namespace, and has been noted in other file system studies [5, 43]. Users and applications group files into locations in the namespace that correspond to their semantic meaning (*e. g.*, a common project, such as a source code tree, or similar file types, such as a directory of binary executable files). We earlier found that the workload is not evenly distributed which causes a similar property to exist for timestamps.

To measure spatial locality, we use an attribute value's *locality ratio*: the percent of directories that recursively contain the value, as illustrated in Figure 3.15. A directory recursively contains an attribute value if it or any of its sub-directories contains the value. The figure on the right has a lower locality ratio because the `ext` attribute value `html` is recursively contained in fewer directories. Using recursive accounting allows our analysis to be more broad since it looks an entire sub-trees rather than individual directories. The root directory (*e. g.*, `/`) recursively contains all of the attributes that occur in the file system. Thus, by definition the locality ratio is a super set of the percent of directories that directly contain an attribute value.



(a) **Locality Ratio=54%** (b) **Locality Ratio=38%**

Figure 3.15: **Examples of Locality Ratio.** Directories that recursively contain the ext attribute value `html` are black and gray. The black directories contain the value. The Locality Ratio of ext value `html` is 54% ($= 7/13$) in the first tree and 38% ($= 5/13$) in the second tree. The value of `html` has better spatial locality in the second tree than in the first one.

Table 3.8 shows the locality ratios for the 32 most frequently occurring values for various attributes (ext, size, owner, ctime, mtime) in each of the traces. Locality ratios are less than 1% for all attributes, meaning that over 99% of directories do not recursively contain the value. We expect extended attributes to exhibit similar properties since they are often tied to file type and owner attributes. **Observation 12:** *Metadata attribute values are heavily clustered in the namespace.*

Utilizing spatial locality can help prune a query's search space by identifying only the parts of the namespace that contain a metadata value. This approach will eliminate a large number of files from the search space. Unfortunately, most general-purpose DBMSs treat pathnames as flat string attributes. As a result, they do not interpret the hierarchical layout of file attributes, making it difficult for them to utilize this information. Instead DBMSs typically must consider *all* files for a search no matter its locality.

3.7.2 Frequency Distribution

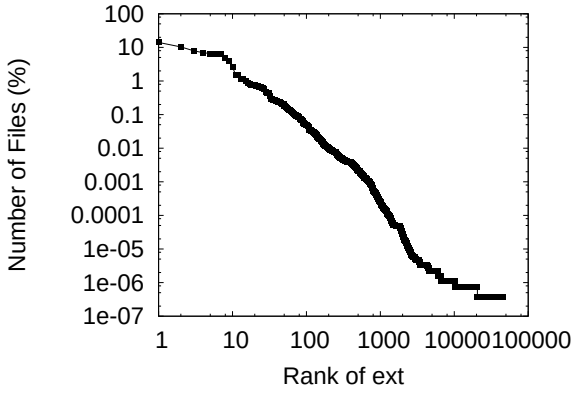
Metadata values also have highly skewed frequencies—their popularity distributions are asymmetric, causing a few very popular metadata values to account for a large fraction of all total values. This distribution has also been observed in other metadata studies [5, 43]. Figures 3.16(a) and 3.16(b) show the distribution of ext and size values from our Home trace on a log-log scale. The linear appearance indicates that the distributions are Zipf-like and follow

	ext	size	uid	ctime	mtime
Web	0.000162% – 0.120%	0.0579% – 0.177%	0.000194% – 0.0558%	0.000291% – 0.0105%	0.000388% – 0.00720%
Eng	0.00101% – 0.264%	0.00194% – 0.462%	0.000578% – 0.137%	0.000453% – 0.0103%	0.000528% – 0.0578%
Home	0.000201% – 0.491%	0.0259% – 0.923%	0.000417% – 0.623%	0.000370% – 0.128%	0.000911% – 0.0103%

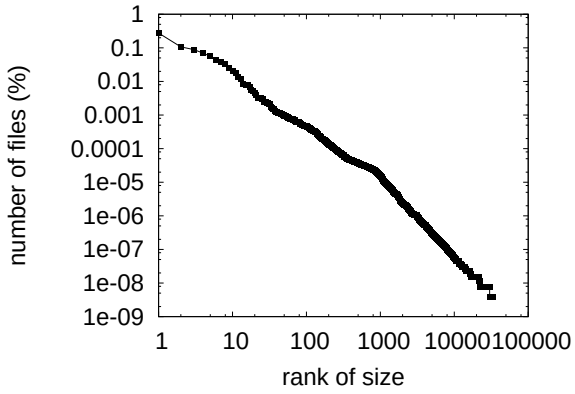
Table 3.8: **Locality Ratios of the 32 most frequently occurring attribute values.** All Locality Ratios are well below 1%, which means that files with these attribute values are recursively contained in less than 1% of directories.

the power law distribution [152]. In these distributions, 80% of files have one of the 20 most popular ext or size values, while the remaining 20% of the files have thousands of other values. Figure 3.16(c) shows the distribution of the Cartesian product (*i. e.*, the intersection) of the top 20 ext and size values. The curve is much flatter, which indicates a more even distribution of values. Only 33% of files have one of the top 20 ext and size combinations. In Figure 3.16(c), file percentages for corresponding ranks are at least an order of magnitude lower than in the other two graphs. This means, for example, that there are many files with owner `andrew` and many files with ext `pdf`, but often there are over an order of magnitude fewer files with *both* owner `andrew` and ext `pdf` attributes. **Observation 13:** *Metadata attribute values have highly skewed frequency distributions.*

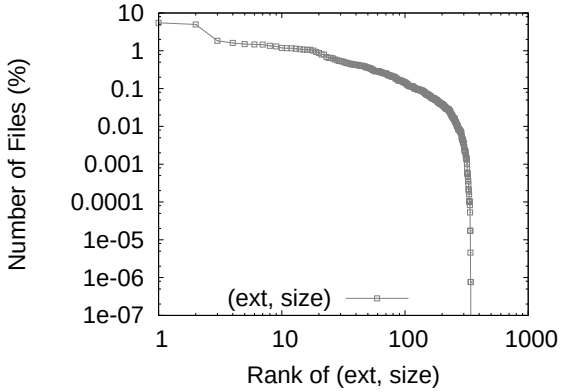
These distribution properties show that multi-attribute searches will significantly reduce the number of query results as Boolean queries return the intersection of the results for each query predicate. Unfortunately, most DBMSs rely on attribute value distributions (also known as selectivity) to choose a query plan. When distributions are skewed, query plans often require extra data processing [101]; for example, they may retrieve all of `andrew`'s files to find the few that are `andrew`'s `pdf` files or vice-versa. Our analysis shows that query execution should utilize attribute values' spatial locality rather than their frequency distributions. Spatial locality provides a more effective way to execute a query because it is more selective and can better reduce a query's search space. Additionally, if frequency distribution is to be used, the frequency of the Cartesian products of the query predicates should be used rather than a single predicate.



(a)



(b)



(c)

Figure 3.16: **Attribute value distribution examples.** A rank of 1 represents the attribute value with the highest file count. The linear curves on the log-log scales in Figures 3.16(a) and 3.16(b) indicate a Zipf-like distribution, while the flatter curve in Figure 3.16(c) indicates a more even distribution.

3.8 Summary

In order to design better file system organization and indexing techniques it is important to understand the kinds of data that file systems stores and how they are used. In this chapter we presented an analysis of *both* file system workload and snapshot traces. We analyzed two large-scale CIFS network file system workloads and three metadata snapshots from enterprise-class file servers.

We compared these workloads to previous file system studies to understand how file access patterns have changed and conducted a number of other experiments. We found that read-write file access patterns and random file access are far more common than previously thought and that most file storage remains unused, even over a three month period. Our observations on sequentiality, file lifetime, file reuse and sharing, and request type distribution also differ from those in earlier studies. Based on these observations, we made several recommendations for improving network file server design to handle the workload of modern corporate and engineering environments. Additionally, we found that metadata attribute values are heavily clustered in the namespace and that attribute values have highly skewed distributions. We then discussed how these attributes impact DBMS performance and how they can be exploited to improve search performance. While our analysis is a first step at designing better organization and indexing strategies there are a number of other important experiments that have yet to be studied. We detail some of these in Section 6.1.

Chapter 4

New Approaches to File Indexing

Achieving effective search in large-scale file systems is difficult because these systems contain petabytes of data and billions of files. Index structures must index up to $10^{10} - 10^{11}$ metadata attributes and even more content keywords. In addition, they must handle frequent file updates. Thus, proper index design is critical to achieving effective file system search.

Unfortunately, current file system search solutions rely on general-purpose index structures that are not designed for file system search and can limit performance at large-scales. As discussed in Section 2.5, metadata attributes are often indexed in a relational DBMS and content keywords are usually indexed in a traditional inverted index. General-purpose index solutions are appealing as off-of-the-shelf solutions that can provide the needed search functionality and are widely available. For example, Microsoft's enterprise search indexes metadata in their Extensible Storage Engine (ESE) DBMS [109]. Also, the Linux desktop search tool, Beagle [18], relies on the standard Lucene inverted index [11] for content search.

While general-purpose index solutions are quick to deploy, they lack the optimized designs required to effectively search billions of files. An effective indexing system must meet several requirements. Search performance must be fast and scalable enough to make it a common method of file access. General-purpose solutions are optimized for other workloads. For example, DBMS are optimized for OLTP workloads, which can cause needless disk accesses, poor cache utilization, and extra processing when used for file system search [162]. Additionally, index update performance must be fast enough to quickly index frequent file changes.

Unfortunately, most general-purpose index structures are optimized for search performance, meaning that updates can be very slow [1, 96]. The index must also have limited resource requirements to ensure that a scalable solution is cost-effective and possible to integrate within an existing storage system. General-purpose indexes often depend on dedicated hardware to ensure performance [15, 75], which makes them expensive at large-scales. While not part of the index itself, it is also critical that file metadata and keywords can be effectively gathered from the file system. It must be possible to quickly collect changes from billions of files without impacting normal file system performance.

In this chapter, we examine the hypothesis that file system search requirements can be better meet through new index designs that are specifically optimized for file systems. To do this we present the design of a file metadata index and a file content index that leverage file system specific properties discussed in the previous chapter to guide their designs and improve performance.

Metadata index: We present the design of Spyglass, a novel metadata search system that exploits file metadata properties to enable fast, scalable search that can be embedded within the storage system. Our design introduces several new metadata indexing techniques. *Hierarchical partitioning* is a new method of namespace-based index partitioning that exploits namespace locality to provide flexible control of the index. *Signature files* are compact descriptions of a partition's contents, helping to route queries only to relevant partitions and prune the search space to improve performance and scalability. A new *snapshot-based* metadata collection method provides scalable collection by re-crawling only the files that have changed. Finally, *partition versioning*, a novel index versioning mechanism, enables fast update performance while allowing "back-in-time" search of past metadata.

Inverted content index: We present an inverted index design that leverages hierarchical partitioning to decompose posting lists into many smaller, disjoint segments based on the file system's namespace. Through the use of an *indirect index* that manages these segments, our approach provides flexible, fine-grained index control that can enhance scalability

and improve both search and update performance. In addition, we discuss how to leverage partitioning to enforce file security permissions, provide personalized search result rankings, and distribute the index across a cluster.

An evaluation of our Spyglass prototype, using our real-world, large-scale metadata traces, shows that search performance is improved 1–4 orders of magnitude compared to basic DBMS setups. Additionally, search performance is scalable; it is capable of searching hundreds of millions of files in less than a second. Index update performance is up to $40\times$ faster than basic DBMS setups and scales linearly with system size. The index itself typically requires less than 0.1% of total disk space. Index versioning allows "back-in-time" metadata search while adding only a tiny overhead to most queries. Finally, our snapshot-based metadata collection mechanism performs $10\times$ faster than a straw-man approach. Our evaluation demonstrates that file system-specific designs can greatly improve performance compared to general-purpose solutions.

This remainder of this chapter is organized as follows. Section 4.1 presents the design of our Spyglass metadata index and Section 4.2 presents the design of our content index. We evaluate performance using our Spyglass prototype in Section 4.3 and summarize our findings in Section 4.4.

4.1 Metadata Indexing

In addition to the file system properties presented in Chapter 3, we wanted to better understand user and administrator metadata search needs. To do this we surveyed over 30 large scale storage system users and administrators. We asked subjects to rank the perceived usefulness of various queries that we supplied, as well as, to supply the kinds of queries they would like to run and why. We found subjects using metadata search for a wide variety of purposes. Use cases included managing storage tiers, tracking legal compliance data, searching large scientific data output files, finding files with incorrect security ACLs, and resource/capacity planning. Table 4.1 provides examples of some popular use cases and the metadata attributes searched.

File Management Question	Metadata Search Query
Which files can be migrated to tape?	size $>$ `50 GB`, atime $>$ `6 months`.
How many duplicates of this file are in my home directory?	owner = `andrew`, datahash = `0xE431`, path = `/home/andrew`.
Where are my recently modified presentations?	owner = `andrew`, type = (`ppt` | `keynote`), mtime $<$ 2 days.
Which legal compliance files can be expired?	retention time = `expired`, mtime $>$ `7 years`
Which of my files grew the most in the past week?	Top 100 where size(today) $>$ size(1 week ago), owner = `andrew`.
How much storage do these users and applications consume?	Sum size where owner = `andrew`, type = `database`

Table 4.1: **Use case examples.** Metadata search use cases collected from our user survey. The high-level questions being addressed are on the left. On the right are the metadata attributes that are being searched and example values. Users used basic inode metadata as well as specialized extended attributes, such as legal retention times. Common search characteristics include multiple attributes, localization to part of the namespace, and "back-in-time" search.

From our survey we observed three important metadata search characteristics. First, over 95% of searches included *multiple metadata attributes* to refine search results; a search on a single attribute over a large file system can return thousands or even millions of results, which users do not want to sift through. The more specific their queries the more narrow the scope of the results. Second, about 33% of *searches were localized* to part of the namespace, such as a home or project directory. Users often have some idea of where their files are and a strong idea of where they are not; localizing the search focuses results on only relevant parts of the namespace. Third, about 25% of the searches that users deemed most important *searched multiple versions* of metadata. Users use "back-in-time" searches to understand file trends and how files are accessed.

4.1.1 Spyglass Design

We designed Spyglass to address the file system search requirements discussed earlier. Spyglass is specially designed to exploit metadata search properties to achieve scale and performance while being embedded within the storage system. Spyglass focuses on crawling, updating, and searching metadata.

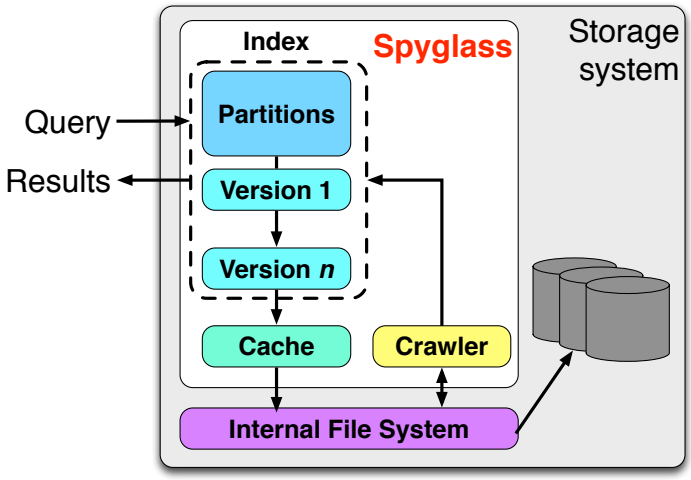


Figure 4.1: **Spyglass overview.** Spyglass resides within the storage system. The crawler extracts file metadata, which gets stored in the index. The index consists of a number of partitions and versions, all of which are managed by a caching system.

Spyglass uses several novel techniques that exploit the file system properties discussed in Chapter 3 to provide fast, scalable search in large-scale storage systems. First, *hierarchical partitioning* partitions the index based on the namespace, preserving spatial locality in the index and allowing fine-grained index control. Second, *signature files* [53] are used improve search performance by leveraging locality to identify only the partitions that are relevant to a query. Third, *partition versioning* versions index updates, which improves update performance and allows "back-in-time" search of past metadata versions. Finally, Spyglass utilizes storage systems snapshots to crawl only the files whose metadata has changed, providing fast collection of metadata changes. Spyglass resides within the storage system and consists of two major components, shown in Figure 4.1: the Spyglass index, which stores metadata and serves queries, and a crawler that extracts metadata from the storage system.

4.1.2 Hierarchical Partitioning

To exploit metadata locality and improve scalability, the Spyglass index is partitioned into a collection of separate, smaller indexes, with a technique we call hierarchical partitioning.

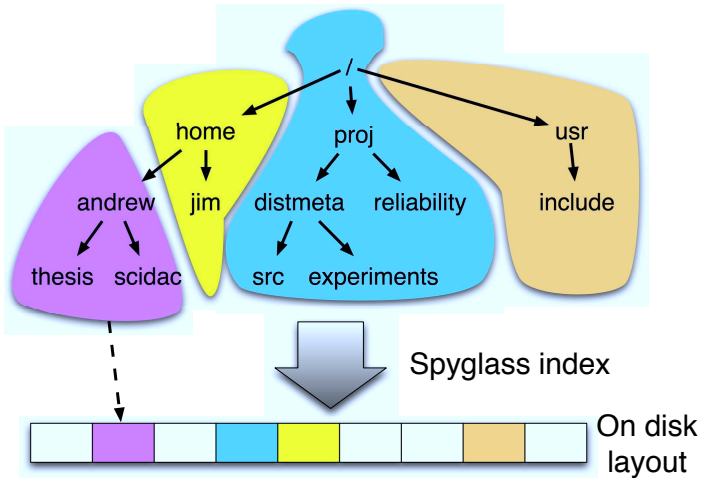


Figure 4.2: **Hierarchical partitioning example.** Sub-tree partitions, shown in different colors, index different storage system sub-trees. Each partition is stored sequentially on disk. The Spyglass index is a tree of sub-tree partitions.

Hierarchical partitioning is based on the storage system's namespace and encapsulates separate parts of the namespace into separate partitions, thus allowing more flexible, fine grained control of the index. Similar partitioning strategies are often used by file systems to distribute the namespace across multiple machines [122, 180].

Each of the Spyglass partitions is stored sequentially on disk, as shown in Figure 4.2. Thus, unlike a DBMS, which stores records adjacently on disk using their row or column order, Spyglass groups records nearby in the namespace together on disk. This approach improves performance since the files that satisfy a query are often clustered in only a portion of the namespace, as shown by our observations in Section 3.7. For example, a search of the storage system for `andrew`'s `ppt` files likely does not require searching sub-trees such as other user's home directories or system file directories. Hierarchical partitioning allows only the sub-trees relevant to a search to be considered, thereby enabling reduction of the search space and improving scalability. Also, a user may choose to localize the search to only a portion of the namespace. Hierarchical partitioning allows users to control the scope of the files that are searched. A DBMS-based solution usually encodes pathnames as flat strings, making it oblivious to the hi-

erarchical nature of file organization and requiring it to consider the entire namespace for each search. If the DBMS stores the files sorted by file name, it can improve locality and reduce the fraction of the index table that must be scanned; however, this approach can still result in performance problems for index updates, and does not encapsulate the hierarchical relationship between files.

Spyglass partitions are kept small, on the order of 100,000 files, to maintain locality in the partition and to ensure that each can be read and searched very quickly. We discuss the reason for choosing this size in Section 4.3. Since partitions are stored sequentially on disk, searches can usually be satisfied with only a few small sequential disk reads to retrieve the partitions that are needed to satisfy a query. Also, sub-trees often grow at a slower rate than the system as a whole [5, 43], which provides scalability because the number of partitions to search will often grow slower than the size of the system.

We refer to each partition as a *sub-tree partition*; the Spyglass index is a tree of sub-tree partitions that reflects the hierarchical ordering of the storage namespace. Each partition has a main *partition index*, in which file metadata for the partition is stored; *partition metadata*, which keeps information about the partition; and pointers to child partitions. Partition metadata includes information used to determine if a partition is relevant to a search and information used to support partition versioning.

The Spyglass index is stored persistently on disk; however, all partition metadata, which is small, is cached in-memory. A *partition cache* manages the movement of entire partition indexes to and from disk as needed. When a file is accessed, its neighbor files will likely need to be accessed as well, due to spatial locality. Paging entire partition indexes allows metadata for all of these files to be fetched in a single, small sequential read. This concept is similar to the use of embedded inodes [57], to store inodes adjacent to their parent directory on disk.

In general, Spyglass search performance is a function of the number of partitions that must be read from disk. Thus, the partition cache's goal is to reduce disk accesses by ensuring that most partitions searched are already in-memory. While we know of no studies of file system query patterns we believe that a simple LRU algorithm is effective. Both web queries [19] and file system access patterns (see Section 3.3) exhibit skewed, Zipf-like popularity distributions,

suggesting that file metadata queries *may* exhibit similar popularity distributions; this would mean that only a small subset of partitions will be frequently accessed. An LRU algorithm keeps these recently accessed partitions in-memory, ensuring high performance for common queries and efficient cache utilization.

4.1.2.1 Partition Indexes

Each partition index must provide fast, multi-dimensional search of the metadata it indexes. To do this we use a K-D tree [24], which is a k-dimensional binary tree, because it provides lightweight, logarithmic point, range, and nearest neighbor search over k dimensions and allows multi-dimensional search of a partition in tens to hundreds of microseconds. However, other index structures can provide additional functionality. For example, FastBit [187] provides high index compression, Berkeley DB [121] provides transactional storage, cache-oblivious B-trees [21] improve B-tree update time, and K-D-B-trees [136] allow partially in-memory K-D trees. However, in most cases, the fast, lightweight nature of K-D trees is preferred. The drawback is that K-D trees are difficult to update; Section 4.1.3 describes techniques to avoid continuous K-D tree updates.

4.1.2.2 Partition Metadata

Partition metadata contains information about the files in the partition, including paths of indexed sub-trees, file statistics, signature files, and version information. File statistics, such as file counts and minimum and maximum values, are kept to answer aggregation and trend queries without having to process the entire partition index. These statistics are computed as files are being indexed. A *version vector*, which is described in Section 4.1.3, manages partition versions. Signature files are used to determine if the partition contains files relevant to a query.

Signature files [53] are bit arrays that serve as compact summaries of a partition's contents and exploit metadata locality to prune a query's search space. A common example of a signature file is the Bloom Filter [26]. Spyglass can determine whether a partition *may* index any files that match a query simply by testing bits in the signature files. A signature file and an associated hashing function are created for each attribute indexed in the partition. All bits in

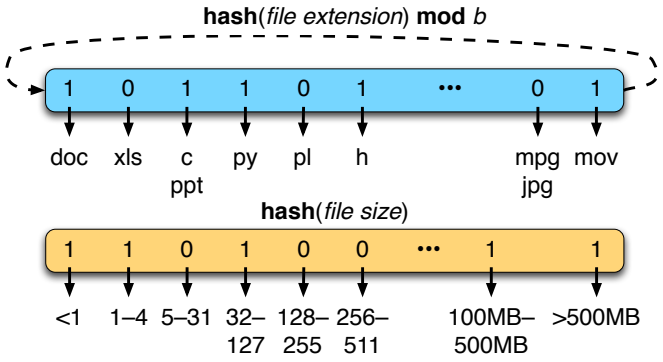


Figure 4.3: **Signature file example.** Signature files for the ext and size metadata attributes are shown. Each bit corresponds to a value or set of values in the attribute value space. One bits indicate that the partition *may* contain files with that attribute value while zero bits that they definitely do not. In the top figure, each bit corresponds to an extension. False-positives occur in cases where multiple extensions hash to the same bit position. In the bottom figure, each bit corresponds to a range of file sizes.

the signature file are initially set to zero. As files are indexed, their attribute values are hashed to a bit position in the attribute's signature file, which is set to one. We illustrate the design of two signature files in Figure 4.3. To determine if the partition indexes files relevant to a query, each attribute value being searched is hashed and its bit position is tested. The partition needs to be searched *only* if *all* bits tested are set to one. Thus, this approach does not depend on the frequency distribution of a single attribute value, which we showed in Section 3.7 is a poor query execution metric. Due to spatial locality, most searches can eliminate many partitions, reducing the number of disk accesses and processing a query must perform.

As a result of collisions in the hashing function that cause false positives, a signature file determines only if a partition *may* contain files relevant to a query, potentially causing a partition to be searched when it does not contain any files relevant to a search. A false positive is shown in the top part of Figure 4.3 where we see both the `c` and `pdf` extensions hash to the same bit location. The one bit at that location can only tell if one of those attributes are stored in the partition, but not which one. However, signature files cannot produce false negatives,

so partitions with relevant files will never be missed. False-positive rates can be reduced by varying the size of the signature or changing the hashing function. Increasing signature file sizes, which are initially around 2 KB, decreases the chances of a collision by increasing the total number of bits. This trades off increased memory requirements and lower false positive rates. Changing the hashing function allow a bit's meaning and how it is used to be improved. For example, consider a signature file for file size attributes, as shown in the bottom part of Figure 4.3. Rather than have each bit represent a single size value (*e. g.*, 522 bytes), we can reduce false positives for common small files by mapping each 1 KB range to a single bit for sizes under 1 MB. The ranges for less common large files can be made more coarse, perhaps using a single bit for sizes between 25 and 50 MB.

While Spyglass stores signature files in memory, it is possible to store them efficiently on disk. Signature files can be written to disk in a *bit-sliced* [54] fashion, which allows only the data for the few bits being tested to be read from disk. Bit-slicing is done by grouping N signature files together, such as the signature files describing the ext attribute for N partitions. These signature files are stored on disk in N-bit slices, where the i^{th} slice contains the i^{th} bit from each of the N signature files. Thus, retrieving slice i during query execution for an ext attribute value will read bit i for the N different signature files from disk. This approach allows the bit in question to be accessed sequentially for N signature files and eliminates the need to read untested bit positions.

When Spyglass contains many partitions, the number of signature files that must be tested can become large. The number of signature files that have to be tested can be reduced by utilizing the tree structure of the Spyglass index to create hierarchically defined signature files. Hierarchical signature files are smaller signatures (roughly 100 bytes) that summarize the contents of its partition and the partitions below it in the tree. Hierarchical signature files are the logical OR of a partition's signature files and the signature files of its children. A single failed test of a hierarchical signature file can eliminate huge parts of the index from the search space, preventing every partition's signature files from being tested. Hierarchical signature files are kept small to save memory at the cost of increased false positives.

4.1.3 Partition Versioning

Spyglass improves update performance and enables "back-in-time" search using a technique called partition versioning that batches index updates, treating each batch as a new incremental index version. The motivation for partition versioning is two-fold. First, we wish to improve index update performance by not having to modify existing index structures. In-place modification of existing indexes can generate large numbers of disk seeks and can cause partition's K-D tree index structure to become unbalanced. Second, back-in-time search can help answer many important storage management questions that can track file trends and how they change.

Spyglass batches updates before they are applied as new versions to the index, meaning that the index may be stale because file modifications are not immediately reflected in the index. However, batching updates improves index update performance by eliminating many small, random, and frequent updates that can thrash the index and cache. Additionally, from our user survey, most queries can be satisfied with a slightly stale index. It should be noted that partition versioning does not require updates to be batched. The index can be updated in real time by versioning each individual file modification, as is done in most versioning file systems [144, 156], however this will increase space requirements and decrease performance.

4.1.3.1 Creating Versions

Spyglass versions each sub-tree partition individually rather than the entire index as a whole in order to maintain locality. A versioned sub-tree partition consists of two components: a *baseline index* and *incremental indexes*, which are illustrated in Figure 4.4. A baseline index is a normal partition index that represents the state of the storage system at time T_0, or the time of the initial update. An incremental index is an index of metadata *changes* between two points in time T_{n-1} and T_n. These changes are indexed in K-D trees, and smaller signature files are created for each incremental index. Storing changes differs from the approach used in some versioning file systems [144], which maintain full copies for each version. VersionFS [115] provides similar semantics to our method by versioning only the deltas between blocks. Changes consist of

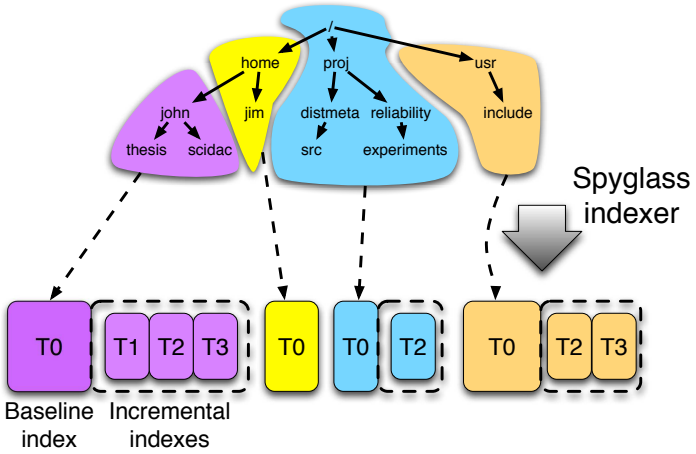


Figure 4.4: **Versioning partitioning example.** Each sub-tree partition manages its own versions. A baseline index is a normal partition index from some initial time T_0. Each incremental index contains the changes required to roll query result forward to a new point in time. Each sub-tree partition manages its version in a version vector.

metadata creations, deletions, and modifications. Maintaining only changes requires a minimal amount of storage overhead, resulting in a smaller footprint and less data to read from disk.

Each sub-tree partition starts with a baseline index, as shown in Figure 4.4. When a batch of metadata changes is received at T_1, it is used to build incremental indexes. Each partition manages its incremental indexes using a *version vector*, similar in concept to inode logs in the Elephant File System [144]. Since file metadata in different parts of the file system change at different rates, as was shown in Section 3.3 and in other studies [5], partitions may have different numbers and sizes of incremental indexes. Incremental indexes are stored sequentially on disk adjacent to their baseline index. As a result, updates are fast because each partition writes its changes in a single, sequential disk access. Incremental indexes are paged into memory whenever the baseline index is accessed, increasing the amount of data that must be read when paging in a partition, though not typically increasing the number of disk seeks. As a result, the overhead of versioning on overall search performance is usually small.

Performing a "back-in-time" search that is accurate as of time T_n works as follows. First, the baseline index is searched, producing query results that are accurate as of T_0. The incremental indexes T_1 through T_n are then searched in chronological order. Each incremental index searched produces metadata changes that modify the search results, rolling them forward in time, and eventually generating results that are accurate as of T_n. For example, consider a query for files with `owner andrew` that matches two files, F_a and F_b, at T_0. A search of incremental indexes at T_1 may yield changes that cause F_b to no longer match the query (*e. g.*, no longer owned by `andrew`), and a later search of incremental indexes at T_n may yield changes that cause file F_c to match the query (*i. e.*, now owned by `andrew`). The results of the query are F_a and F_c, which is accurate as of T_n. Because this process is done in memory and each version is relatively small, searching through incremental indexes is often very fast. In rolling results forward, a small penalty is paid to search the most recent changes; however, updates are much faster because no data needs to be copied, as is the case in CVFS [156], which rolls version changes backwards rather than forwards.

4.1.3.2 Managing Versions

Over time, older versions tends to decrease in value and should be removed to reduce search overhead and save space. Spyglass provides two efficient techniques for managing partition versions: *version collapsing* and *version checkpointing*. Version collapsing applies each partition's incremental index changes to its baseline index. The result is a single baseline for each partition that is accurate as of the most recent incremental index. Collapsing is efficient because all original index data is read sequentially and the new baseline is written sequentially. During collapsing the signature files are re-computed to remove one bits that may correspond to attribute values that no longer exist. An extensible hashing [98] method may be used to incrementally grow or shrink the signature files if needed. Version checkpointing allows an index to be saved to disk as a new copy to preserve an important landmark version of the index and is similar to file landmarks in Elephant [144].

We describe how collapsing and checkpointing can be used with an example. During the day, Spyglass is updated hourly, creating new versions every hour, thus allowing "back-in-

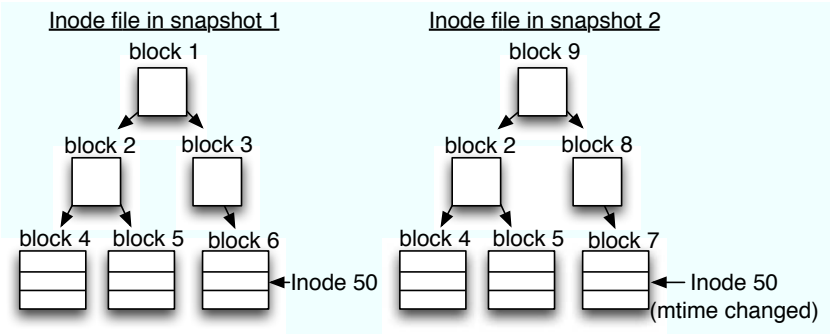


Figure 4.5: **Snapshot-based metadata collection.** In snapshot 2, block 7 has changed since snapshot 1. This change is propagated up the tree. Because block 2 has not changed, we do not need to examine it or any blocks below it.

time" searches to be performed at per-hour granularity over the day. At the end of each day, incremental versions are collapsed, reducing space overhead at the cost of prohibiting hour-by-hour searching over the last day. Also, at the end of each day, a copy of the collapsed index is checkpointed to disk, representing the storage system state at the end of each day. At the end of each week, all but the latest daily checkpoints are deleted; and at the end of each month, all but the latest weekly checkpoints are deleted. This results in versions of varying time scales. For example, over the past day any hour can be searched, over the past week any day can be searched, and over the past month any week can be searched. The frequency for index collapsing and checkpointing can be configured based on user needs and space constraints.

4.1.4 Collecting Metadata Changes

The Spyglass crawler takes advantage of NetApp Snapshot™ technology in the NetApp WAFL ® file system [77] on which it was developed to quickly collect metadata changes. Given two snapshots, T_{n-1} and T_n, Spyglass calculates the difference between them. This difference represents all of the file metadata changes between T_{n-1} and T_n. Because of the way snapshots are created, only the metadata of *changed* files is re-crawled. This approach is faster than current approaches which often re-crawl all or most of the file system.

All metadata in WAFL resides in a single file called the *inode file*, which is a collection of fixed length inodes. Extended attributes are included in the inodes. Performing an initial crawl of the storage system is fast because it simply involves sequentially reading the inode file. Snapshots are created by making a copy-on-write clone of the inode file. Calculating the difference between two snapshots leverages this mechanism. This is shown in Figure 4.5. By looking at the block numbers of the inode file's indirect and data blocks, we can determine exactly which blocks have changed. If a block's number has not changed, then it does not need to be crawled. If this block is an indirect block, then no blocks that it points to need to be crawled either because block changes will propagate all the way back up to the inode file's root block. As a result, the Spyglass crawler can identify just the data blocks that have changed and crawl only their data. This approach greatly enhances scalability because crawl performance is a function of the number of files that have changed rather than the total number of files.

Spyglass is not dependent on snapshot-based crawling, though it provides benefits compared to alternative approaches. Periodically walking the file system can be extremely slow because each file must be traversed. Moreover, traversal can utilize significant system resources and alter file access times on which file caches depend. Another approach, file system event notifications (*e. g.*, `inotify` [86]), requires hooks into critical code paths, potentially impacting performance. A change log, such as the one used in NTFS, is another alternative; however, since we are not interested in every system event, a snapshot-based scheme is more efficient.

4.1.5 Distributed Design

Our discussion thus far has focused on indexing and crawling on a single storage server. However, large-scale storage systems are often composed of tens or hundreds of servers. While we do not currently address how to distribute the index, we believe that hierarchical partitioning lends itself well to a distributed environment because the Spyglass index is a tree of partitions. A distributed file system with a single namespace can view Spyglass as a larger tree composed of partitions placed on multiple servers. As a result, distributing the index is a matter of effectively scaling the Spyglass index tree. Also, the use of signature files may be

effective at routing distributed queries to relevant servers and their sub-trees. Obviously, there are many challenges to actually implementing this. A complete distributed design is one of the future directions for this work that we discuss in Section 6.2.

4.2 Content Indexing

In this section we look at how similar file system specific indexing techniques can be applied to file content search. As mentioned previously, the inverted index is the chief data structure for keyword search. We outline how these indexing techniques can be applied to an inverted index. In particular, our inverted index design utilizes *hierarchical partitioning*, which we introduced in Section 4.1 and which exploits namespace locality. Namespace locality implies that a file's location within the namespace influences its properties. Part of our design is based upon the assumption that keyword distributions exhibit a similar namespace locality property as metadata. That is, the content keywords are also clustered in the namespace. For example, file's containing the keyword "financial", "budget", and "shareholder" are more likely to be contained in directories pertaining to a company's quarterly financial documents rather than a developer source code tree or a directory of system files.

While most evidence for file system content keywords exhibiting namespace locality is anecdotal, a previous study did find a variety of keywords that were more common in some file system data sets than others [167]. A more complete file system keyword analysis is future work discussed in Section 6.2. However, even in the absence of keyword namespace locality, our approach provides a method for fine grained control of the index and posting lists that users can use to localize their searches, improve update performance, and which can utilize other partitioning strategies, such as partitioning along file security permissions.

4.2.1 Index Design

Recall from Section 2.6.2 that an inverted index contains a dictionary of keywords that map to posting lists, which specify the locations in the file system where the keywords occur. Our new index design consists of two-levels. At the first level is a single inverted index,

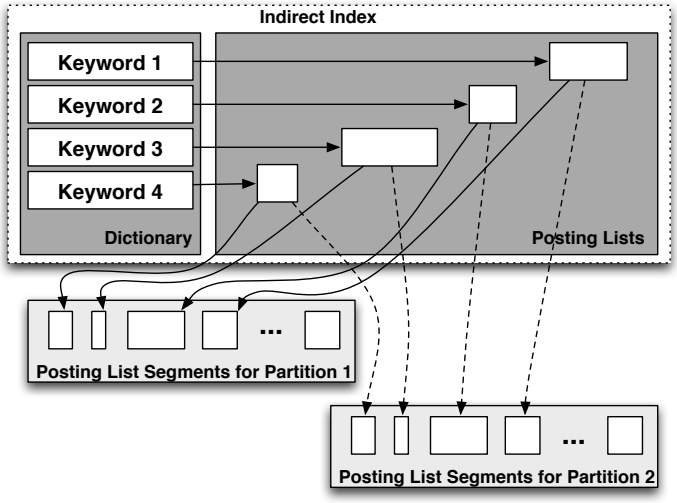


Figure 4.6: **Indirect Index Design.** The indirect index stores the dictionary for the entire file system and each keyword's posting lists contain locations of partition segments. Each partition segment is kept sequential on-disk.

called the *indirect index*, that points to the locations of posting list *segments* rather than the entire posting list itself and is illustrated in Figure 4.6. The indirect index is similar to the inverted index used in GLIMPSE [104]. At the second level is a large collection of posting list segments. A posting list segment is a region of a posting list that is stored sequentially on-disk. Posting lists are partitioned into segments using hierarchical partitioning. Thus, a segment represents the postings for a keyword that occurs within a specific sub-tree in the namespace. An illustration of how a posting list is partitioning into segments is shown in Figure 4.7. The namespace is partitioned so that each sub-tree's partition is relatively small, on the order of 100,000 files, similar to our design in Spyglass. By partitioning the posting lists into segments we ensure fast performance for searching or updating any one partition, as posting lists are small enough to efficiently read, write, and cache in-memory. In essence, partitioning makes the index namespace locality-aware and allow the index to be controlled at the granularity of sub-trees. However, it should be pointed out the partition does not need to be based on namespace location and other metrics, such as security or owner, may also be appropriate.

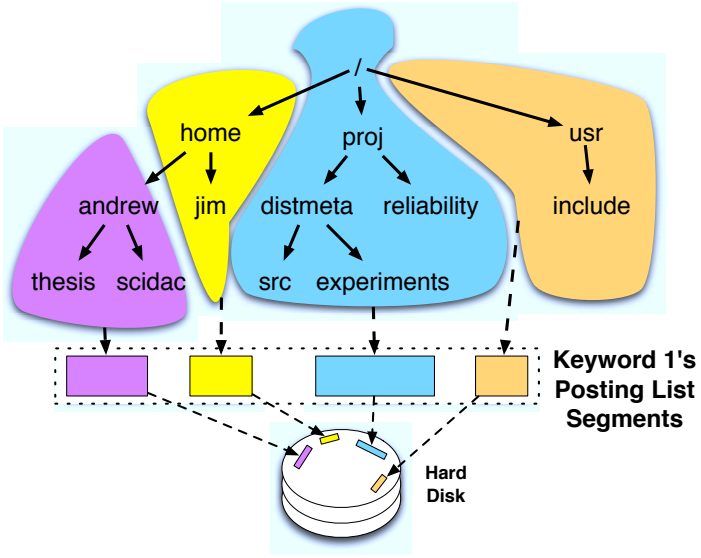


Figure 4.7: **Segment Partitioning.** The namespace is broken into partitions that represent disjoint sub-trees. Each partition maintains posting list segments for keywords that occur within its sub-trees. Since each partition is relatively small, these segments can be kept sequential on-disk.

The purpose of the indirect index is to identify which sub-tree partitions contain any postings for a given keyword. Doing so allows search, update, security, and ranking to operate at the granularity of sub-trees. The indirect index maintains the dictionary for the entire file system. The reason to maintain a single dictionary is that keeping a dictionary per-partition would simply require too much space overhead since many keywords will be replicated in many dictionaries. Each keyword's dictionary entry points to a posting list that contains the on-disk address of segments that contain actual postings, which is shown in Figure 4.6. Since the indirect index only maintains a dictionary and posting lists containing segment pointers, it can be kept in-memory if properly distributed across the nodes in the file system, which we will discuss later in this chapter.

4.2.2 Query Execution

All search queries go through the indirect index. The indirect index identifies which segments contain the posting data relevant to the search. Since each segment is sequential on-disk, retrieving a single segment is fast. A disk seek will often be required between segments.

While retrieving a keyword's full posting list (*i. e.*, all segments for the keyword) requires a disk seek between each segment, our use of hierarchical partitioning allows us to exploit namespace locality to retrieve fewer segments. As mentioned earlier, it is assumed that keywords and phrases have namespace locality and only occur in a fraction of the partitions (which we plan to quantify in future our future work). For example, the Boolean query $storage \wedge research \wedge santa \wedge cruz$ requires (depending on the ranking algorithm) that a partition contain files with all four terms before it should be searched. If it does not contain all four terms, often it does not need to be searched at all. Using the indirect index, we can easily identify the partitions that contain the full *intersection* of the query terms. By taking the intersection of the partitions returned, we can identify just the segments that contain files matching the query. Reading only these small segments can significantly reduce the amount of data read compared to fetching postings from across the entire file systems. Likewise, by reducing the search space to a few small partitions, with disk seeks occurring along partition boundaries, the total number of disk seeks can be significantly reduced.

The search space can also be reduced when a search query is localized to part of the namespace. For example, a user may want to search only their home directory or the sub-tree containing files for a certain project. In existing systems, the entire file system is searched and then results are pruned to ensure they fall within the sub-tree. However, through the use of a look up table that maps directory pathnames to their partitions, our approach reduces the scope of the search space to only the scope specified in the query. For example, a query scoped to a user's home directory eliminates all segments not within their home directory from the search space. Thus, users can control the scope and performance of their queries, which is critical in large-scale file systems. Often as the file system grows, the files a user cares about searching

and accessing grows at a much slower rate. Our approach allows search to scale with what the user wants to search, rather than the total size of the file system.

Once in-memory, segments are managed by an LRU cache. As mentioned previously, there have been no studies of file system query patterns though web searches [19, 94] and file access patterns(see Section 3.3) both exhibit Zipf-like distributions. This implies skewed popularity distributions are likely for partitions and that an LRU algorithm will be able to keep popular partitions in-memory, greatly improving performance for common-case searches. Additionally, this enables better cache utilization since only index data related to popular partitions is kept in-memory, rather than data from all over the file system. Efficient cache utilization is important for direct integration with the file system since it will often share the same hardware with the file system.

4.2.3 Index Updates

One of the key challenges with file system search is balancing search and update performance. As discussed in Section 2.6.2, inverted indexes traditionally use either an in-place or merge-based update strategy [96]. An in-place update strategy is an update-optimized approach. When postings lists are written to disk, a sequential region on-disk is allocated that is larger than the required amount. When new postings are added to the list they are written to the empty region. However, when the region fills and new posting needs to be written, a new sequential region is allocated elsewhere on-disk and the new postings are written to it. Thus, in-place updates are fast to write since they can usually be written sequentially and do not require much pre-processing. However, as posting lists grow they become very fragmented which degrades search performance. Alternatively, a merge-based update strategy is a search-optimized approach. When a posting list is modified it is read from disk, modified in-memory, and written out sequentially to a new location. This strategy ensures that posting lists are sequential on-disk, though requires the entire posting to be read and written in order to update it, which can be extremely slow for large posting lists.

Our approach achieves a better balance in two ways. First, since posting list segments only contain postings from partitions, they are small enough to make merge-based updates

efficient. When modifying a posting list, we are able to quickly read the entire list, modify it in memory, and quickly write it out sequentially to disk. Doing so keeps segment updates relatively fast and ensures that segments are sequential on-disk. An in-place approach is also feasible since small segments often will not need to allocate more than one disk block though the space overhead from over-allocating disk blocks for many segments can become quite high. Second, our approach can exploit locality in file access patterns to reduce overall disk I/Os. Often only a subset of file system sub-trees are frequently modified as we showed in Section 3.3.6 and was shown in others studies [5, 43]. As a result, queries often only need to read segments from a small number of partitions. By reading fewer segments, far less data needs to read for an update compared to retrieving an entire posting list, cache space is better utilized, and updates can be coalesced in-memory before being written back to disk.

4.2.4 Additional Functionality

In addition to improving scalability, hierarchical partitioning can potentially improve how file permissions are enforced, aid result ranking, and improve space utilization.

Secure file search is difficult because either an index is kept for each user, which requires a huge amount of disk space, or permissions for all search results need to be checked, which can be very slow [33]. However, most users only have access privileges to a limited number of sub-trees in the namespace [125]. Hierarchical partitioning, through the use of additional security metadata stored in the indirect index, can eliminate sub-trees a user cannot access from the search space. Doing so prevents users from searching files they cannot access without requiring any additional indexes and reduces the total number of search results returned, which limits the number of files whose permissions must be checked.

Ranking file system search results is difficult because most files are unstructured documents with little semantic information. However, sub-trees in the namespace often have distinct, unique purposes, such as a user's home directory or a source code tree. Using hierarchical partitioning, we can leverage this information to improve how search results are ranked in two ways. First, files in the same partition may have a semantic relationship (*i. e.*, files for the same project) that can be used when calculating rankings. Second, different ranking requirements

may be set for different partitions. Rather than use a "one size fits all" ranking function for all billion files in the file system, we can potentially use different ranking functions for different parts of the namespace. For example, files from a source code tree can be ranked differently than files in a scientist's test data, potentially improving search relevance for users.

Both file system access patterns and web searches have Zipf-like distributions. Assuming these distributions hold true for file system search, a large set of index partitions will be cold (*i. e.*, not frequently searched). Our approach can allow us to identify these cold partitions and either heavily compress them or migrate them to lower-tier storage (low-end disk or tape) to improve cost and space utilization. A similar concept has been applied to legal compliance data in file systems and has shown the potential for significant space savings [112].

4.2.5 Distributing the Index

Large-scale inverted indexes are usually distributed [15] as are large-scale file systems. A distributed design enables better scalability and parallel processing capabilities. It is important that the index is aware of how the file system is distributed so that it can place index components near the data that they index and can effectively balance load across the nodes. We now discuss how our index can be distributed across the nodes in a file system. We use parallel file systems, such as Ceph [180], as the context for our discussion because they are intended for large-scale environments and are highly distributed. In a parallel file system, a cluster of metadata servers (MDSs) handles metadata operations while a cluster of object storage devices (OSDs) handles data operations. The MDS cluster manages the namespace and stores all file metadata persistently on the OSD cluster. A more in-depth discussion of parallel file systems is provided in Section 2.2.

We intend for the indirect index to be distributed across the MDS cluster and across enough nodes so that it can be kept in-memory. Since a significant amount of query preprocessing takes place in the indirect index (*e. g.*, determining which partitions to search), keeping it in-memory will significantly improve performance, however is not required. Posting list segments will be stored on the OSD cluster and since they are small they can map directly to

physical objects. Storing segments on the OSD cluster also provides parallel access to many segments at a time.

The indirect index will be partitioned across the MDS cluster using a *global inverted file* (IF_G) partitioning approach [15]. In this approach keywords are used to partition the index such that each MDS stores only a subset of the keywords in the file system. For example, with two MDS nodes A and B, A may index and store all keywords in the range $[a - q]$ and B may index and store all remaining keywords. Along with a good keyword partitioning strategy, IF_G can provide good load balancing and limit network bandwidth requirements as messages are sent only to the MDS nodes responsible for keywords in the query.

In our design, the example Boolean query $storage \wedge santa \wedge cruz$ will be evaluated as follows. A user will issue the query through a single MDS node (possibly of their choosing) which will shepherd the query execution. This shepherd node will query the MDS nodes responsible for the keywords "storage", "santa", and "cruz" based on the IF_G partitioning. These nodes will return their indirect index posting lists, which are stored in-memory, and the shepherd will compute the intersection of these to determine which partitions contain all three terms and are thus relevant to the query. The shepherd will cache these posting lists (to improve subsequent query performance) and then query the other MDS nodes for the segments that correspond to the relevant partitions. These segments will be read from the OSD cluster, cached at the three MDS nodes, and returned to the shepherd. The shepherd will aggregate the results list from the segments and rank them before returning them to the user.

4.3 Experimental Evaluation

In this section we evaluate how well the new indexing techniques we presented address the file system search challenges described in the beginning of the chapter and how our designs compare to general-purpose solutions. To do this, we evaluate a Spyglass prototype implementation. Due to the lack of a representative file system content keyword data set and the need to first examine whether namespace locality impacts keywords in large-scale file systems, we do not evaluate our inverted index design. However, since our metadata and content index

designs share many features, the results of our Spyglass evaluation are still very relevant. To evaluate Spyglass we first measured metadata collection speed, index update performance, and disk space usage. We then analyzed search performance and how effectively index locality is utilized. Finally, we measured partition versioning overhead.

4.3.1 Implementation Details

Our Spyglass prototype was implemented as a user-space process on Linux. An RPC-based interface to WAFL gathers metadata changes using our snapshot-based crawler. Our prototype dynamically partitions the index as it is being updated. As files and directories are inserted into the index, they are placed into the partition with the longest pathname match (*i. e.*, the pathname match farthest down the tree). New partitions are created when a directory is inserted and all matching partitions are full. A partition is considered full when it contains over 100,000 files. We use 100,000 as the soft partition limit in order to ensure that partitions are small enough to be efficiently read and written to disk. Using a much smaller partition size will often increase the number of partitions that must be accessed for a query; this incurs extra expensive disk seeks. Using a much larger partition size decreases the number of partitions that must be accessed for a query; however it poorly encapsulates spatial locality, causing extra data to be read from disk. In the case of symbolic and hard links, multiple index entries are used for the file.

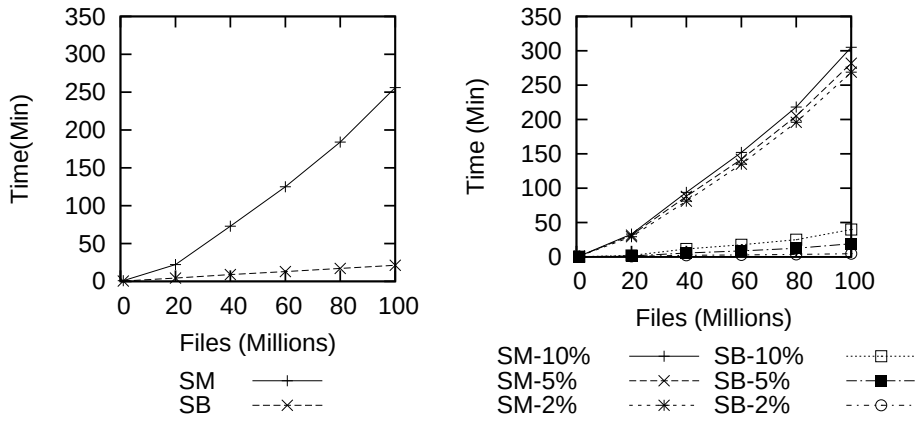
During the update process, partitions are buffered in-memory and written sequentially to disk when full; each is stored in a separate file. K-D trees were implemented using `libkdtree++` [97]. Signature file bit-arrays are about 2 KB, but *hierarchical* signature files are only 100 bytes, ensuring that signature files can fit within our memory constraints. Hashing functions that allowed each signature file's bit to correspond to a range of values were used for file size and time attributes to reduce false positive rates. When incremental indexes are created, they are appended to their partition on disk. Finally, we implement a simple search API that allows point, range, top-k, and aggregation searches. We plan to extend this interface as future work.

4.3.2 Experimental Setup

We evaluated performance using our metadata snapshot traces described in Table 3.6. These traces have varying sizes, allowing us to examine scalability. Our Web and Eng traces also have incremental snapshot traces of daily metadata changes for several days. Since no standard metadata search benchmarks exist, we constructed synthetic sets of queries, discussed later in this section, from our metadata traces to evaluate search performance. All experiments were performed on a dual core AMD Opteron machine with 8 GB of main memory running Ubuntu Linux 7.10. All index files were stored on a network partition that accessed a high-end NetApp file server over NFS.

We also evaluated the performance of two popular relational DBMSs, PostgreSQL and MySQL, which serve as relative comparison points to DBMS-based solutions used in other metadata search systems. The goal of our comparison is to provide some context to frame our Spyglass evaluation, not to compare performance to the best possible DBMS setup. We compared Spyglass to an index-only DBMS setup, which is used in several commercial metadata search systems, and also tuned various options, such as page size, to the best of our ability. This setup is effective at pointing out several basic DBMS performance problems. DBMS performance *can* be improved through the techniques discussed in Chapter 2; however, as stated earlier, they do not completely match metadata search cost and performance requirements.

Our Spyglass prototype indexes the metadata attributes listed in Table 3.7. Our index-only DBMSs include a base relation with the same metadata attributes and a B+-tree index for each. Each B+-tree indexes table row ID. An index-only design reduces space usage compared to some more advanced setups, though it has slower search performance. In all three traces, cache sizes were configured to 128 MB, 512 MB, and 2.5 GB for the Web, Eng, and Home traces, respectively. These sizes are small relative to the size of their trace and correspond to about 1 MB for every 125,000 files, which provides linear scaling of cache sizes.



(a) Baseline

(b) Incremental: 2%, 5%, and 10% changes from baseline

Figure 4.8: **Metadata collection performance.** We compare Spyglass's snapshot-based crawler (SB) to a straw-man design (SM). Our crawler has good scalability; performance is a function of the number of changed files rather than system size.

4.3.3 Metadata Collection Performance

We first evaluated our snapshot-based metadata crawler and compared it to a straw-man approach. Fast collection performance impacts how often updates occur and system resource utilization. Our straw-man approach performs a parallelized walk of the file system using `stat()` to extract metadata. Figure 4.8(a) shows the performance of a baseline crawl of all file metadata. Our snapshot based crawler is up to $10\times$ faster than our straw-man for 100 million files because our approach simply scans the inode file. As a result, a 100 million file system is crawled in less than 20 minutes.

Figure 4.8(b) shows the time required to collect incremental metadata changes. We examine systems with 2%, 5%, and 10% of their files changed. For example, a baseline of 40 million files and 5% change has 2 million changed files. For the 100 million file tests, each of our crawls finishes in under 45 minutes, while our straw-man takes up to 5 hours. Our crawler is able to crawl the inode file at about 70,000 files per second. Our crawler effectively scales

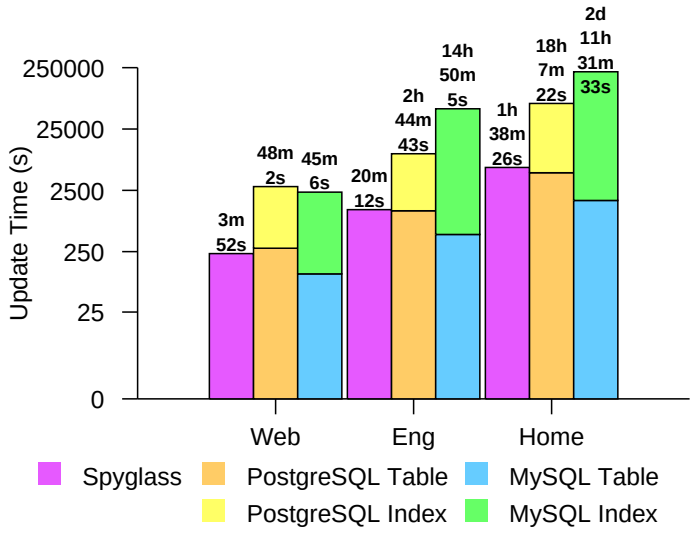


Figure 4.9: **Update performance.** The time required to build an initial baseline index shown on a log-scale. Spyglass updates quickly and scales linearly because updates are written to disk mostly sequentially.

because we incur only a fractional overhead as more files are crawled; this is due to our crawling only changed blocks of the inode file.

4.3.4 Update Performance

Figure 4.9 shows the time required to build the initial index for each of our metadata traces. Spyglass requires about 4 minutes, 20 minutes, and 100 minutes for the three traces, respectively. These times correspond to a rate of about 65,000 files per second, indicating that update performance scales linearly. Linear scaling occurs because updates to each partition are written sequentially, with seeks occurring only between partitions. Incremental index updates have a similar performance profile because metadata changes are written in the same fashion and few disk seeks are added. Our reference DBMSs take between $8\times$ and $44\times$ longer to update because DBMSs require loading their base table and updating index structures. While loading the table is fast, updating index structures often requires seeks back to the base table or extra

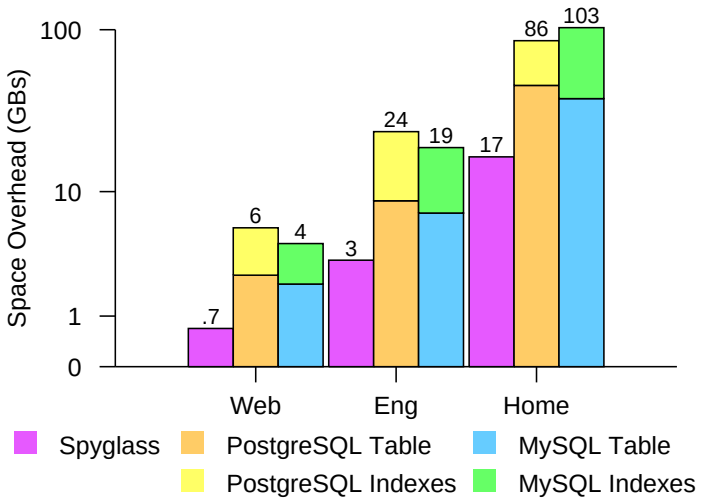


Figure 4.10: **Space overhead.** The index disk space requirements shown on a log-scale. Spyglass requires just 0.1% of the Web and Home traces and 10% of the Eng trace to store the index.

data copies. As a result, DBMS updates with our Home trace can take a day or more; however, approaches such as cache-oblivious B-trees [21] may be able to reduce this gap.

4.3.5 Space Overhead

Figure 4.10 shows the disk space usage for all three of our traces. Efficient space usage has two primary benefits: less disk space taken from the storage system and the ability to cache a higher fraction of the index. Spyglass requires less than 0.1% of the total disk space for the Web and Home traces. However, it requires about 10% for the Eng trace because the total system size is low due to very small files. Spyglass requires about 50 bytes per file across all traces, resulting in space usage that scales linearly with system size. Space usage in Spyglass is $5\times$–$8\times$ lower than in our references DBMSs because they require space to store the base table and index structures. Figure 4.10 shows that building index structures can more the double the total space requirements.

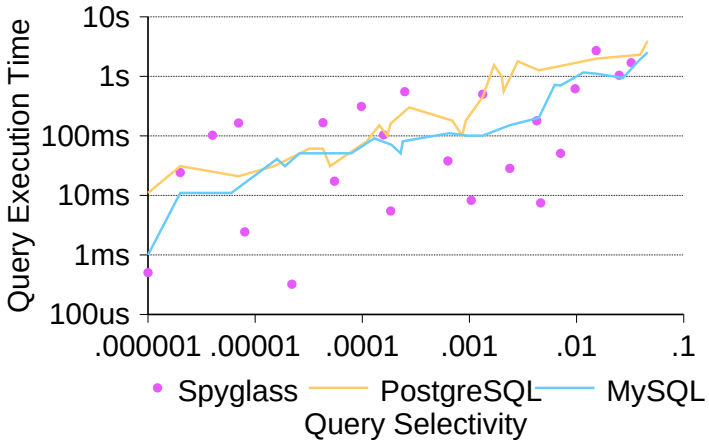


Figure 4.11: **Comparison of Selectivity Impact.** The selectivity of queries in our query set is plotted against the execution time for that query. We find that query performance in Spyglass is much less correlated to the selectivity of the query predicates than the DBMSs, which are closely correlated with selectivity.

4.3.6 Selectivity Impact

We evaluated the effect of metadata selectivity on the performance of Spyglass and the DBMSs. We again generated query sets of ext and owner from the Web trace with varying *selectivity*—the ratio of the number of results to all records. Figure 4.11 plots query selectivity against query execution time. We found that the performance of PostgreSQL and MySQL is highly correlated with query selectivity. However, this correlation is much weaker in Spyglass, which exhibits much more variance. For example, a Spyglass query with selectivity 7×10^{-6} runs in 161 ms while another with selectivity 8×10^{-6} requires 3 ms. This variance is caused by the higher sensitivity of Spyglass to hierarchical locality and query locality, as opposed to simple query selectivity. This behavior is unlike that of a DBMS, which accesses records from disk based on the predicate it thinks is the most selective. The weak correlation with selectivity in Spyglass means it is less affected by the highly skewed distribution of storage metadata which makes determining selectivity difficult.

Set	Search	Metadata Attributes
Set 1	Which user and application files consume the most space?	Sum sizes for files using owner and ext.
Set 2	How much space, in this part of the system, do files from query 1 consume?	Use query 1 with an additional directory path.
Set 3	What are the recently modified application files in my home directory?	Retrieve all files using mtime, owner, ext, and path.

Table 4.2: **Query Sets.** A summary of the searches used to generate our evaluation query sets.

4.3.7 Search Performance

To evaluate Spyglass search performance, we generated sets of queries derived from real-world queries in our user study; there are, unfortunately, no standard benchmarks for file system search. These query sets are summarized in Table 4.2. Our first set is based on a storage administrator searching for the user and application files that are consuming the most space (*e. g.*, total size of `andrew`'s `vmdk` files)—an example of a simple two-attribute search. The second set is an administrator localizing the same search to only part of the namespace, which shows how localizing the search changes performance. The third set is a storage user searching for recently modified files of a particular type in a specific sub-tree, demonstrating how searching many attributes impacts performance. Each query set consists of 100 queries, with attribute values randomly selected from our traces. Randomly selecting attribute values means that our query sets loosely follow the distribution of values in our traces and that a variety of values are used.

Figure 4.12 shows the total run times for each set of queries. In general, query set 1 takes Spyglass the longest to complete, while query sets 2 and 3 finish much faster. This performance difference is caused by the ability of sets 2 and 3 to localize the search to only a part of the namespace by including a path with the query. Spyglass is able to search only files from this part of the storage system by using hierarchical partitioning. As a result, the search space for these queries is bound to the size of the sub-tree, no matter how large the storage system. Because the search space is already small, using many attributes has little impact on performance for set 3. Query set 1, on the other hand, must consider all partitions and tests each partition's signature files to determine which to search. While many partitions are eliminated,

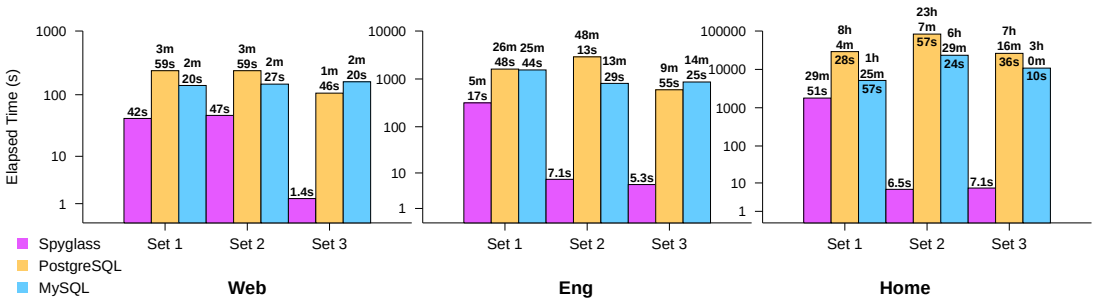


Figure 4.12: **Query set run times.** The total time required to run each set of queries. Each set is labeled 1 through 3 and is clustered by trace file. Each trace is shown on a separate log-scale axis. Spyglass improves performance by reducing the search space to a small number of partitions, especially for query sets 2 and 3, which are localized to only a part of the namespace.

there are more partitions to search than in the other query sets, which accounts for the longer run times.

Our comparison DBMSs perform closer to Spyglass on our smallest trace, Web; however, we see the gap widen as the system size increases. In fact, Spyglass is over four orders of magnitude faster for query sets 2 and 3 on our Home trace, which is our largest at 300 million files. The large performance gap is due to several reasons. First, our DBMSs consider files from all parts of the namespace, making the search space much larger. Second, skewed attribute value distributions cause our DBMSs to process extra data even when there are few results. Third, the DBMSs base tables ignore metadata locality, causing extra disk seeks to find files close in the namespace but far apart in the table. Spyglass, on the other hand, uses hierarchical partitioning to significantly reduce the search space, performs only small, sequential disk accesses, and can exploit locality in the workload to greatly improve cache utilization.

Using the results from Figure 4.12, we calculated query throughput, shown in Table 4.3. Query throughput (queries per second) provides a normalized view of our results and the query loads that can be achieved. Spyglass achieves throughput of multiple queries per second in all but two cases; in contrast, the reference DBMSs do not achieve one query per second in any instance, and, in many cases, cannot even sustain one query per five minutes. Figure 4.13 shows an alternate view of performance; a cumulative distribution function (CDF) of query ex-

System	Web			Eng			Home		
	Set 1	Set 2	Set 3	Set 1	Set 2	Set 3	Set 1	Set 2	Set 3
Spyglass	2.38	2.12	71.4	0.315	14.1	18.9	0.05	15.4	14.1
PostgreSQL	0.418	0.418	0.94	0.062	0.034	0.168	0.003	0.001	0.003
MySQL	0.714	0.68	0.063	0.647	0.123	0.115	0.019	0.004	0.009

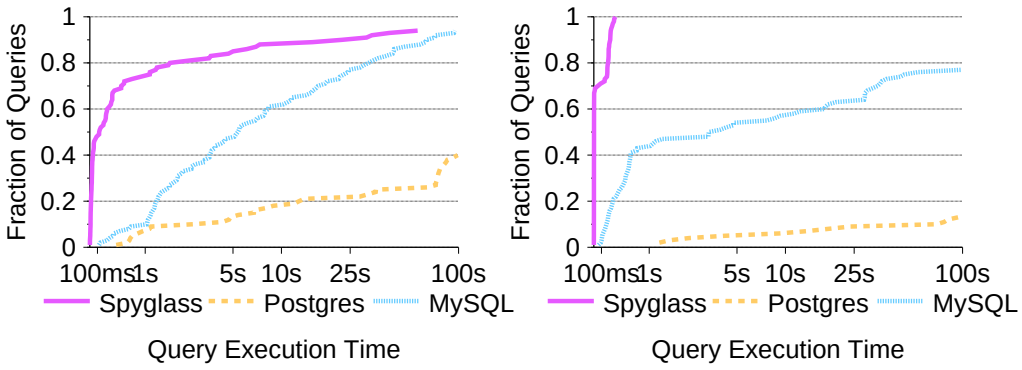
Table 4.3: **Query throughput.** We use the results from Figure 4.12 to calculate query throughput (queries per second). We find that Spyglass can achieve query throughput that enables fast metadata search even on large-scale storage systems.

ecution times on our Home trace, allowing us to see how each query performed. In query sets 2 and 3, Spyglass finishes all searches in less than a second because localized searches bound the search space. For query set 1, we see that 75% of queries take less than one second, indicating that most queries are fast and that a few slow queries contribute significantly to the total run times in Figure 4.12. These queries take longer because they must read many partitions from disk, either because few were previously cached or many partitions are searched.

4.3.8 Index Locality

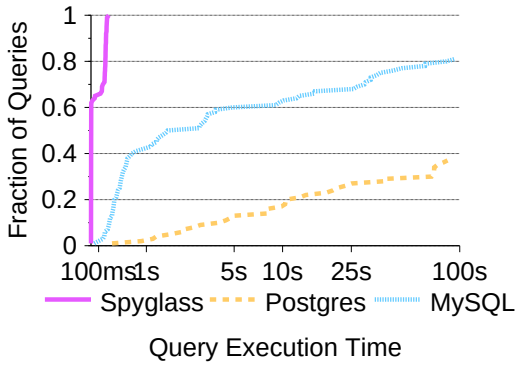
We now evaluate how well Spyglass exploits spatial locality to improve query performance. We generated another set of queries, based on query 1 from Table 4.2, with 500 queries with owner and ext values randomly selected from our Eng trace. We generated similar query sets for individual ext and owner attributes.

Figure 4.14(a) shows a CDF of the fraction of partitions searched. Searching more partitions often increases the amount of data that must be read from disk, which decreases performance. We see that 50% of searches using just the ext attribute reference fewer than 75% of partitions. However, 50% of searches using both ext and owner together reference fewer than 2% of the partitions, since searching more attributes increases the locality of the search, thereby reducing the number of partitions that must be searched. Figure 4.14(b) shows a CDF



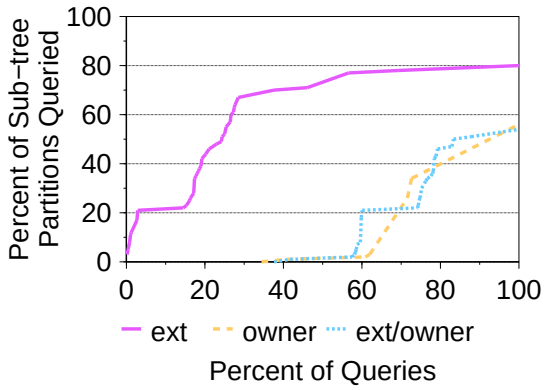
(a) Query set 1.

(b) Query set 2.

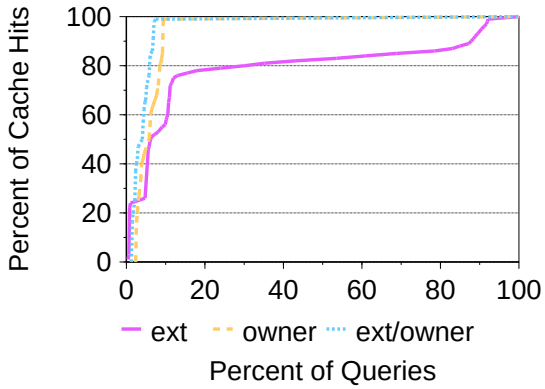


(c) Query set 3.

Figure 4.13: **Query execution times.** A CDF of query set execution times for the Eng trace. In Figures 4.13(b) and 4.13(c), all queries are extremely fast because these sets include a path predicate that allows Spyglass to narrow the search to a few partitions.



(a) CDF of sub-tree partition accesses.



(b) CDF of partition cache hits.

Figure 4.14: **Index locality.** A CDF of the number of partitions accessed and the number of accesses that were cache hits for our query set. Searching multiple attributes reduces the number of partition accesses and increases cache hits.

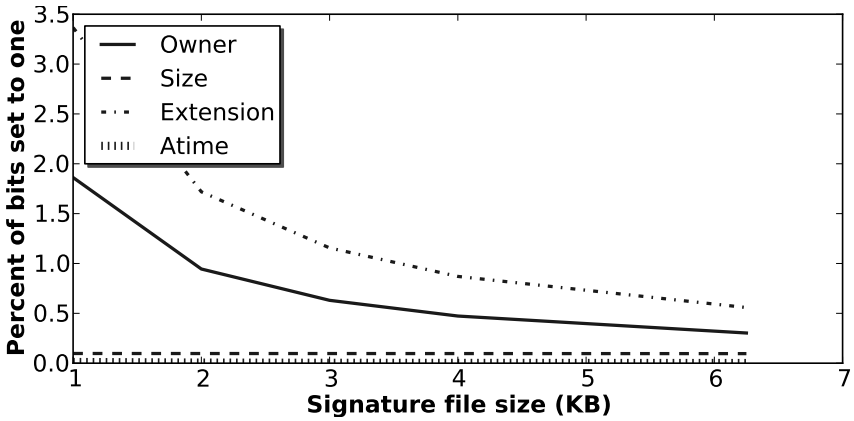


Figure 4.15: **Impact of signature file size.** The percent of one bits in signature files for four different attributes is shown. The percent of one bits decreases as signature file size increases. However, even in the case where signature files are only 1 KB less than 4% of total bits are set to one. As mentioned earlier, the size and atime attributes use special hashing functions that treat each bit as a range rather than a discrete value. This approach keeps the fraction of one bits below 1% in all cases.

of cache hit percentages for the same set of queries. Higher cache hit percentages means that fewer partitions are read from disk. Searching owner and ext attributes together results in 95% of queries having a cache hit percentage of 95% or better due to the higher locality exhibited by multi-attribute searches. The higher locality causes repeated searches in the sub-trees where these files reside and allows Spyglass to ignore more non-relevant partitions.

The number of partitions searched during a query is dependent on how effectively signature files can correctly eliminate partitions from the search space. Signature files that are small will often produce more false-positives, which causes the number of partitions searched to increase. The probability of a false-positive is dependant on the percent of bits in the signature file that are set to one [26]. Figure 4.15 shows how the percent of one bits change with signature file size for four different attributes in Spyglass using our Web trace. The average percent of one bits across all of the partitions for each of the attributes is shown. We see that the percent of one bits decreases as the size of the signature files increase. For the ext and owner attributes, the percent is $2-3\times$ lower for 6 KB signature files, compared to 1 KB. However, it is important

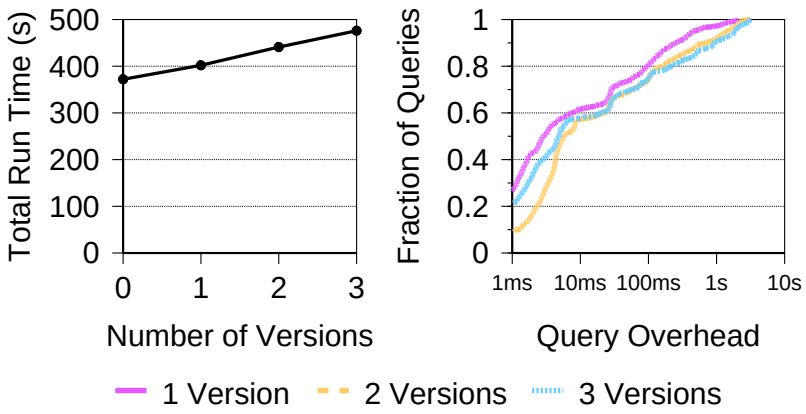


Figure 4.16: **Versioning overhead.** The figure on the left shows total run time for a set of 450 queries. Each version adds about 10% overhead. On the right, a CDF shows per-query overheads. Over 50% of queries have an overhead of 5 ms or less.

to point out that even with signature files at 1 KB, the total percent of bits set to one is less than 4%. The reason for this is that hierarchical partitioning exploits namespace locality to keep files with similar attribute values together. As a result, each partition only has a limited number of attribute values. Also, the size and atime signature files use hashing functions where each bit corresponds to a range rather than a discrete value, as described in Section 4.1.2. As a result, the percent of one bits is always well below 1%. Our Spyglass prototype uses 2 KB signature files for each of the ten attributes described in Table 3.7. This size yields good accuracy and is compact, requiring only 200 MB of memory for 1 billion files.

4.3.9 Versioning Overhead

To measure the search overhead added by partition versioning, we generated 450 queries based on query 1 from Table 4.2 with values randomly selected from our Web trace. We included three full days of incremental metadata changes, and used them to perform three incremental index updates. Figure 4.16 shows the time required to run our query set with an increasing number of versions; each version adds about a 10% overhead to the total run time. However, the overhead added to most queries is quite small. Figure 4.16 also shows, via a

CDF of the query overheads incurred for each version, that more than 50% of the queries have less than a 5 ms overhead. Thus, it is a few much slower queries that contribute to most of the 10% overhead. This behavior occurs because overhead is typically incurred when incremental indexes are read from disk, which doesn't occur once a partition is cached. Since reading extra versions does not typically incur extra disk seeks, the overhead for the slower queries is mostly due to reading partitions with much larger incremental indexes from disk.

4.4 Summary

Providing effective search at the scale of billions of files is not easy. There are a number of requirements, such as scalable file crawling, fast search and update performance, and efficient resource utilization, that must be addressed. The design of an effective search index is critical to meeting these requirements. However, existing search solutions rely on general-purpose index structures, such as DBMSs, that are not designs for file search and which limit performance and scalability.

In this chapter we examined the hypothesis that these requirements can be better meet with index designs that are optimized for file system search. To examine this hypothesis we presented the design of two new indexing structures; one for file metadata and one for content search. Unlike general-purpose indexes, our index designs leveraged the large-scale file system properties presented in Chapter 3 to improve performance and scalability. Our designs introduced several novel file system indexing techniques. Flexible index control is provided by an index partitioning mechanism, called hierarchical partitioning, that leverages namespace locality. We introduced the use of signature files and an indirect index to effectively route queries and significantly reduce a query's search space. A novel index versioning mechanism was used to provide both fast index updates and "back-in-time" search. An evaluation of our metadata index shows search performance improvements up to 1–4 orders of magnitude compared to existing DBMS based solutions, while providing faster update performance and using only a fraction of the disk space. Our findings support a similar hypothesis from the DBMS community that argues application-specific designs will often out perform a general-purpose solution.

Chapter 5

Towards Searchable File Systems

Search is becoming an increasingly important way for users to access and manage their files. However, current file systems are ill-suited to meet these emerging needs because they organize files using a basic hierarchical namespace that is not easy to search. Modern file organizations still resemble those designed over forty years ago, when file systems contained orders of magnitude fewer files and basic hierarchical namespace navigation was more than sufficient [38]. As a result, searching a file system requires brute-force namespace traversal, which is not practical at large scale. Currently to address this problem, file search is implemented with a search application—a separate index or database of the file system's attributes and metadata outside of the file system—as is done in Linux (*e. g.*, the `locate` tool), personal computers [14], and enterprise search appliances [67, 85].

Though search applications have been somewhat effective for desktop and small-scale servers, they face several inherent limitations at larger scales. First, search applications must track all file changes in the file system, a difficult challenge in a system with billions of files and constant file changes. Second, file changes must be quickly re-indexed to prevent a search from returning very old and inaccurate results. Keeping the application's index and file system consistent is difficult because collecting file changes is often slow [81, 158] and search applications are often inefficient to update [1, 173]. Third, search applications often require significant disk, memory, and CPU resources to manage larger file systems using the same techniques that

are successful at smaller scales. Thus, a new approach is necessary to scale file system search to large-scale file systems.

An alternative solution is to build file search functionality directly into the file system. This eliminates the need to manage a secondary database, allowing file changes to be searched in real-time, and enabling internal file organization that corresponds to the users' need for search functionality. However, enabling search within the file system has its own challenges. First, file metadata and data must be organized and indexed so that it can be searched quickly, even as the system scales. Second, this organization must still provide good file system performance. Previous approaches, such as replacing the file system with a relational database [59, 119], have had difficulty addressing these challenges.

In this chapter we explore the hypothesis that search can be integrated directly into the file system's design to enable scalable and efficient search while providing good file system performance. To examine this hypothesis we present two new approaches to how file systems internally organize and index files. The first is Magellan, a searchable metadata architecture and the second is Copernicus, a semantic file system design that organizes files into a dynamic, search-based namespace.

Magellan: Unlike previous work, Magellan does not use relational databases to enable search. Instead, it uses new query-optimized metadata layout, indexing, and update techniques to ensure searchability and high performance in a single file system. Users view a traditional hierarchical interface, though in Magellan, *all* metadata and file look ups, including directory look ups, are handled using a single search structure, eliminating the redundant data structures that plague existing file systems with search grafted on. Our evaluation of Magellan shows that it is possible to provide a scalable, fast, and searchable metadata system for large-scale storage, thus facilitating file system search without hampering performance.

Copernicus: Unlike Magellan, which enables search withing a traditional hierarchical namespace, Copernicus enables search to be directly integrated into a dynamic, search-based namespace. Copernicus uses a dynamic graph-based index that clusters semantically re-

lated files into vertexes and allows inter-file relationships to form edges between them. This architecture significantly differs from previous semantic file system designs which simply impose a naming layer over a standard file system or database. This graph replaces the traditional directory hierarchy, can be efficiently queried, and allows the construction of dynamic namespaces. The namespace allows "virtual" directories that correspond to a query and allows navigation using inter-file relationships.

The remainder of this chapter is organized as follows. Section 5.1 discusses the challenges for combining search and file systems. The Magellan design is presented in Section 5.2 and the Copernicus design is presented in Section 5.3. Our Magellan prototype implementation is evaluated in Section 5.4. We summarize our findings in Section 5.5.

5.1 Background

Hierarchical file systems have long been the "standard" mechanism for accessing file systems, large and small. As file systems have grown in both size and number of files, however, the need for file search has grown; this need has not been adequately met by existing approaches.

5.1.1 Search Applications

File system search is traditionally addressed with a separate search application, such as the Linux `locate` program, Apple Spotlight [14] and Google Enterprise Search [67]. Search applications re-index file metadata and file content in a separate search-optimized structure, often a relational database or information retrieval engine. These applications augment the file system, providing the ability to efficiently search files without the need for file system modifications, making them easy to deploy onto existing systems. Figure 5.1 shows how these applications interact with the file system. The application maintains search indexes for file metadata and content, such as databases or inverted files, which are stored persistently as files in the file system.

These applications have been somewhat successful on desktop and smaller scale file systems. However, they require that two separate indexes of all metadata be maintained—

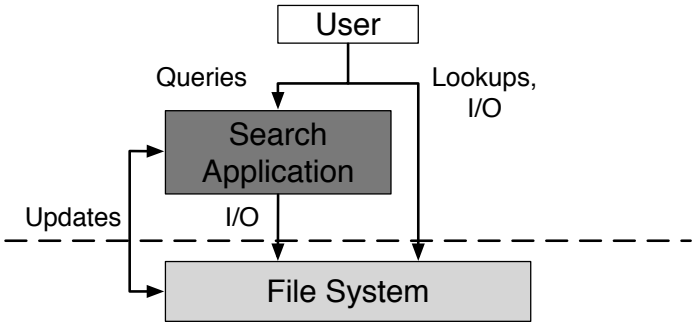


Figure 5.1: **File search applications.** The search application resides on top of the file system and stores file metadata and content in separate search-optimized indexes. Maintaining several large index structures can add significant space and time overheads.

both the file system's index and the search application's index—which presents several inherent challenges as a large-scale and long-term solution:

1) *File attributes and changes must be replicated in the search application.* Metadata and content is replicated into the search appliance by *pulling* it from the file system or having it *pushed* in by the file system. A pull approach, as used by Google Enterprise, discovers file changes through periodic crawls of the file system. These crawls are slow in file systems containing tens of millions to billions of files that must be crawled. Worse, the file system's performance is usually disrupted during the crawl because of the I/O demands imposed by a complete file system traversal. Crawls cannot collect changes in real-time, which often leads to inconsistency between the search application and file system, thus causing incorrect (out-of-date) search results to be returned.

Pushing updates from the file system into the application allows real-time updates. However, the file system must be modified to be aware of the search application. Additionally, search applications are search-optimized, which often makes update performance notoriously slow [1, 173]. Apple Spotlight, which uses a push approach, does not apply updates in real-time for precisely these reasons. As a result, searches in Apple Spotlight may not reflect the most recent changes in the file system, though such files are often the ones desired by the user.

2) *Search applications consume additional resources.* Search applications often rely on abundant hardware resources to enable fast performance. For example, many commercial products can only be purchased as a hardware and software bundle [67, 85]. Requiring additional hardware resources makes these systems expensive and difficult to manage at large-scales. Modern large file systems focus on energy consumption and consolidation [9], making efficient resource utilization critical.

3) *Search appliances add a level of indirection.* Building databases on top of file systems has inefficiencies that have been known for decades [160]; thus, accessing a file through a search application can be much less efficient than through a file system [148]. Accessing a file requires the search application to query its index to find the matching files, which will often require accessing index files stored in the file system. Once file names are returned, the file names are copied to the file system and the files are then retrieved from the file system itself, which requires navigating the file system's namespace index for each file. Accessing files found through searches in a search application require at least double the number of steps, which is both inefficient and redundant.

4) *Users must interact with two interfaces.* Accessing files requires users to interact with two different file interfaces depending on how they want to retrieve their data. The application's query interface must be used for file search while normal file access is performed through the file system's standard interface (*e. g.*, POSIX). Using multiple interfaces to achieve a common goal is both cumbersome and complicates interactions with the storage system.

5.1.2 Integrating Search into the File System

We believe that a more complete solution is for the file system to organize files in a way that facilitates efficient search. Search applications and file systems share the same goal: organizing and retrieving files. Implementing the two functions separately leads to duplicate functionality and inefficiencies. With search becoming an increasingly common way to access and manage files, file systems must provide this functionality as an integral part of their functionality. However, organizing file system metadata so that it can efficiently be searched is not an easy task.

Because current file systems provide only basic directory tree navigation, search applications are the *only* option for flexible, non-hierarchical access. The primary reason behind this shortcoming is that, despite drastic changes in technology and usage, metadata designs remain similar to those developed over 40 years ago [38], when file systems held less than 10 MB. These designs make metadata search difficult for several reasons.

1) *Queries must quickly access large amounts of data from disk.* File systems often have metadata scattered throughout the disk [57]. Thus, scanning the metadata of millions of files for a search can require many expensive disk seeks. Moreover, scanning file content for a search (*e. g.*, `grep`) may read most of the disk's contents, which is not only slow, especially given the rapidly increasing disk capacities compared to slower improvements in disk bandwidth, but file data can also be highly scattered on disk [153].

2) *Queries must quickly analyze large amounts of data.* File metadata and data must be linearly scanned to find files that match the query. File systems do not directly index file metadata or content keywords, which forces slow linear search techniques to be used to find relevant files.

3) *File systems do not know where to look for files.* The file system does not know where the relevant files for a query are located and must often search a large portion of (or the entire) the file system. In large-scale systems, searching large parts of the file system can be impractical because of the sheer volume of data that must be examined.

5.2 Integrating Metadata Search

In this section we present the design of a searchable metadata architecture for large-scale file systems called Magellan. We designed Magellan with two primary goals. First, we wanted a metadata organization that could be quickly searched. Second, we wanted to provide the same metadata performance and reliability that users have come to expect in other high performance file systems. We focus on the problems that make current designs difficult to search, leaving other useful metadata designs intact. Our design leverages metadata specific indexing techniques we developed in Chapter 4.

This section discusses the new metadata techniques that Magellan uses to achieve these goals:

- The use of a search-optimized metadata layout that clusters the metadata for a sub-tree in the namespace on disk to allow large amounts of metadata to be quickly accessed for a query.
- Indexing metadata in multi-dimensional search trees that can quickly answer metadata queries.
- Efficient routing of queries to particular sub-trees of the file system using Bloom filters [26].
- The use of metadata journaling to provide good update performance and reliability for our search-optimized designs.

Magellan was designed to be the metadata server (MDS) for Ceph, a prototype large-scale parallel file system [180]. In Ceph, metadata is managed by a separate metadata server outside of the data path. We discuss issues specific to Ceph where necessary, though our design is applicable to many file systems; systems such as PVFS [34] use separate metadata servers, and an optimized metadata system can be integrated into standard Linux file systems via the `vfs` layer, since Magellan's interface is similar to POSIX though with the addition of a query interface.

5.2.1 Metadata Clustering

In existing file systems, searches must read large amounts of metadata from disk since searching the file system require traversing the directory tree and may need to perform millions of `readdir()` and `stat()` operations to access file and directory metadata. For example, a search to find where a virtual machine has saved a user's virtual disk images may read all metadata below `/usr/` to find files with owner equal to `3407` (the user's UID) and file type equal to `vmdk`.

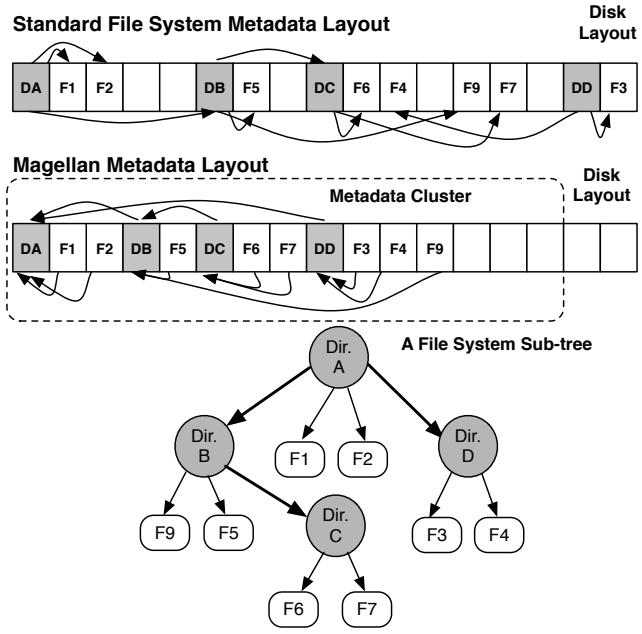


Figure 5.2: **Metadata clustering.** Each block corresponds to an inode on disk. Shaded blocks labeled 'D' are directory inodes while non-shaded blocks labeled 'F' are file inodes. In the top disk layout, the indirection between directory and file inodes causes them to be scattered across the disk. The bottom disk layout shows how metadata clustering co-locates inodes for an entire sub-tree on disk to improve search performance. Inodes reference their parent directory in Magellan; thus, the pointers are reversed.

Accessing metadata often requires numerous disk seeks to access the file and directory inodes, limiting search performance. Though file systems attempt to locate inodes near their parent directory on disk, inodes can still be scattered across the disk. For example, FFS stores inodes in the same on disk cylinder group as their parent directory [105]. However, prior work has shown that inodes for a directory are often spread across multiple disk blocks. Furthermore, directory inodes are not usually adjacent to the first file inode they name, nor are file inodes often adjacent to the next named inode in the directory [57]. We illustrate this concept in the top part of Figure 5.2, which shows how a sub-tree can be scattered on disk.

Magellan addresses this problem by grouping inodes into large groups called *clusters*, which are similar in concept to hierarchical partitioning which was presented in Section 4.1.2. Each cluster contains the metadata for a sub-tree in the namespace and is stored sequentially in a serialized form on disk, allowing it to be quickly accessed for a query. For example, a cluster may store inodes corresponding to the files and directories in the `/projects/magellan/` sub-tree. The bottom part of Figure 5.2 shows how clusters are organized on disk. Retrieving all of the metadata in this sub-tree can be done in a single large sequential disk access. Conceptually, metadata clustering is similar to embedded inodes [57] which store file inodes adjacent to their parent directory on disk. Metadata clustering goes further and stores a group of file inodes and directories adjacent on disk. Co-locating directories and files makes hard links difficult to implement. We address this with a table that tracks hard linked files whose inodes are located in another cluster.

Metadata clustering exploits several file system properties. First, disks are much better at sequential transfers than random accesses. Metadata clustering leverages this to prefetch an entire sub-tree in a single large sequential access. Second, file metadata exhibits *namespace locality*: Metadata attributes are dependent on their location in the namespace, which we showed in Section 3.7. For example, files owned by a certain user are likely to be clustered in that user's home directory or their active project directories, not randomly scattered across the file system. Thus, queries will often need to search files and directories that are nearby in the namespace. Clustering allows this metadata to be accessed more quickly using fewer I/O requests. Third, metadata clustering works well for many file system workloads that exhibit similar locality in their workloads, as was shown in Section 3.3 and prior studies [137]. Often, workloads access multiple, related directories, which clustering works well for.

5.2.1.1 Cluster organization

Clusters are organized into a hierarchy, with each cluster maintaining pointers to its *child clusters*—clusters containing sub-trees in the namespace. A simple example is a cluster storing inodes for `/usr/` and `/usr/lib/` and pointing to a child cluster that stores inodes for `/usr/include/` and `/usr/bin/`, each of which points to its own children. This hierarchy

can be navigated in the same way as a normal directory tree. Techniques for indexing inodes within a cluster are discussed in Section 5.2.2.

While clustering can improve performance by allowing fast prefetching of metadata for a query, it can negatively impact performance if it becomes too large, since clusters that are too large waste disk bandwidth by prefetching metadata for unneeded files. Magellan prevents clusters from becoming too large by using a hard limit on the number of directories a cluster can contain and a soft limit on the number of files. While a hard limit on the number of directories can be enforced by splitting clusters with too many directories, we chose a soft limit on files to allow each file to remain in the same cluster as its parent directory. Our evaluation found that clusters with tens of thousands of files provide the best performance, as discussed in Section 5.4.

5.2.1.2 Creating and caching clusters

Magellan uses a greedy algorithm to cluster metadata. When an inode is created, it is assigned to the cluster containing its parent directory. File inodes are always placed in this cluster. If the new inode is a directory inode, and the cluster has reached its size limit, a new cluster is created as a child of the current directory and the inode is inserted into it. Otherwise, it is inserted into the current cluster. Though this approach works fairly well in practice, it does have drawbacks. First, a very large directory will result in a very large cluster. Second, no effort is made to achieve a uniform distribution of cluster sizes. These issues can be addressed with a clustering algorithm that re-balances cluster distributions over time.

Allocation of a new cluster is similar to allocation of extents in extent-based file systems, such as XFS [168]. When a cluster is created, a sequential region is allocated on disk with a size that is a function of the preset maximum cluster size (generally on the order of 1 to 2 MBs). This allocated region is intended to be larger than the size of the cluster to allow the cluster to grow over time without having to be relocated and without causing fragmentation. Since Magellan is being design for use in Ceph, it is beneficial if allocated regions can correspond to a single Ceph object because it eases cluster management. These regions are dynamically allocated because in Ceph free space and object allocation are managed using a pseudo-random hashing function [181]. However, it is also possible to use an extent tree, as is

used in XFS. With this hashing function, the location of an object is calculated using its object identifier and maps to a Ceph storage device on which the object is stored. The storage device manages its own internal allocation and storage of objects. In the case where a cluster grows too large for a single object it can become fragmented across objects and slow access times. However, Ceph provides the opportunity for parallel access to objects that are on separate storage devices.

Magellan manages memory using a *cluster cache* that is responsible for paging clusters to and from disk, using a basic LRU algorithm to determine which clusters to keep in the cache. Clusters can be flushed to disk under five conditions: (1) the cluster cache is full and needs to free up space; (2) a cluster has been dirty for too long; (3) there are too many journal entries and a cluster must be flushed to free up journal space (as discussed in Section 5.2.4); (4) an application has requested that the cluster be flushed (*e. g.*, via `sync()`); or (5) it is being flushed by a background thread that periodically flushes clusters to keep the number of dirty clusters low. Clusters index inodes using in-memory search trees that cannot be partially paged in or out of memory, so the cache is managed in large, cluster-sized units. The cluster cache will write the cluster to its previous on disk location provided there is enough space. If not, the cluster will written to a new location that is large enough and the old space will be freed. In Ceph, clusters larger than a single object may be striped across multiple storage devices to provide parallel access.

5.2.2 Indexing Metadata

Searches must quickly analyze large amounts of metadata to find the files matching a query. However, current file systems do not index the metadata attributes that need to be searched. For example, searching for files with owner equal to UID `3047` and modification time earlier than `7 days ago`, requires linearly scanning every inode because it is not known which may match the query.

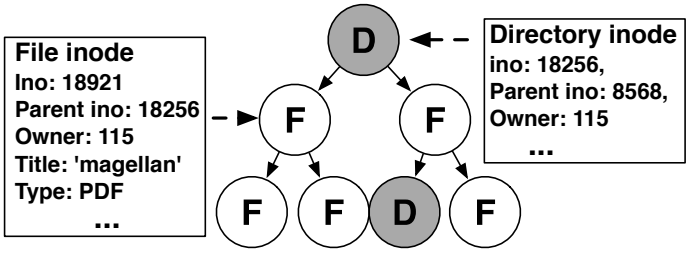


Figure 5.3: **Inode indexing with a K-D tree.** A K-D tree is shown with nodes that are directory inodes are shaded with a 'D'. File inodes are not shaded and labeled 'F'. K-D trees are organized based on attribute value not namespace hierarchy. Thus, a file inode can point to other file inodes, *etc.* The namespace hierarchy is maintained by inodes containing the inode number of their parent directory. Extended attributes, such a title and file type are included in the inode.

5.2.2.1 Indexing with K-D trees

To address this problem, each cluster indexes its inodes in a *K-D tree* : a k-dimensional binary search tree [24]. Inode metadata attributes (*e. g.*, owner, size) are dimensions in the tree and any combination of these can be searched using point, range, or nearest neighbor queries. K-D trees are similar to binary trees, though different dimensions are used to pivot at different levels in the tree. K-D trees allow a single data structure to index all of a cluster's metadata. A one-dimensional data structure, such as a B-tree, would require an index for each attribute, making reading, querying, and updating metadata more difficult.

Each inode is a node in the K-D tree, and contains basic attributes and any extended attributes (*e. g.*, file type, last backup date, *etc.*) that are indexed in the tree, as shown in Figure 5.3. Figure 5.3 shows that inodes are organized based on their attribute values, not their order in the namespace. For example, a file inode's right pointer points to another file inode because it has a lower value for some attribute. It is important to note that inodes often store information not indexed by the K-D tree, such as block pointers.

To maintain namespace relationships, each inode stores its own name and the inode number of its parent directory, as shown in Figure 5.3. A `readdir()` operation simply queries the directory's cluster for all files with parent inode equal to the directory's inode number.

Storing names with their inodes allows queries for filename to not have to locate the parent directory first and eliminates the level of indirection between directories and their entry inodes that exist in normal file systems. In fact, *all* file system metadata operations translate to K-D tree queries. For example, an `open()` on `/usr/foo.txt` is a query in the cluster containing `/usr/`'s inode for a file with filename equal to `foo.txt`, parent inode equal to `/usr/`'s inode number, and with the appropriate mode permissions.

5.2.2.2 Index updates

As a search-optimized data structure, K-D trees provide the best search performance when they are balanced. However, adding or removing nodes can make it less balanced. Metadata modifications must do both: remove the old inode and insert an updated one. While updates are fast $O(\log N)$ operations, many updates can unbalance the K-D tree. The cluster cache addresses this problem by rebalancing a K-D trees before it is written to disk. Doing so piggybacks the $O(N \log N)$ cost of rebalancing onto the bandwidth-limited serialization back to disk, hiding the delay. This approach also ensures that, when a K-D tree is read from disk, it is already optimized.

5.2.2.3 Caching inodes

While K-D trees are good for multi-attribute queries, they are less efficient for some common operations. Many file systems, such as Apple's HFS+ [13], index inodes using just the inode number, often in a B-tree. Operations such as path resolution that perform look ups using just an inode number are done more efficiently in a B-tree than a K-D tree that indexes multiple attributes, since searching just one dimension in a K-D tree uses a range query that requires $O(kN^{1-1/k})$ time, where k is the number of dimensions and N is the size of the tree as compared to a B-tree's $O(\log N)$ look up.

To address this issue, each cluster maintains an *inode cache* that stores pointers to inodes previously accessed in the K-D tree. The inode cache is a hash table that short-circuits inode look ups, avoiding K-D tree look ups for recently-used inodes. The inode cache uses

filename and parent inode as the keys, and is managed by an LRU algorithm. The cache is not persistent; it is cleared when the cluster is evicted from memory.

5.2.3 Query Execution

Searching through many millions of files is a daunting task, even when metadata is effectively clustered out on disk and indexed. Fortunately, as we mentioned earlier in Section 5.2.1, metadata attributes exhibit namespace locality, which means that attribute values are influenced by their namespace location and files with similar attributes are often clustered in the namespace.

Magellan exploits this property by using *Bloom filters* [26] to describe the contents of each cluster and to *route queries* to only the sub-trees that contain relevant metadata. Bloom filters serve a similar role as signature files in our Spyglass design from Chapter 4. Each cluster stores a Bloom filter for each attribute type that it indexes. Bits in the Bloom filters are initialized to zero when they are created. As inodes are inserted into the cluster, metadata values are hashed to positions in the bit array, which are set to one. In a Bloom filter, a one bit indicates that a file with that attribute *may* be indexed in the cluster, while a zero bit indicates that the cluster contains no files with that attribute. A one bit is probabilistic because of hash collisions; two attribute values may hash to the same bit position causing false-positives. A query only searches a cluster when *all* bits tested by the query are set to one, eliminating many clusters from the search space. False positives cause a query to search clusters that do not contain relevant files, degrading performance but not leading to incorrect results. Magellan keeps Bloom filters small (a few kilobytes) to ensure that they fit in memory.

Unfortunately, deleting values from Bloom filters is difficult, since when removing or modifying an attribute, the bit corresponding to the old attribute value cannot be set to zero because the cluster may contain other values that hash to that bit position. However, not deleting values will cause false positives to increase. To address this, Magellan clears and recomputes Bloom filters when a cluster's K-D tree is being flushed to disk. Writing the K-D tree to disk visits each inode, allowing the Bloom filter to be rebuilt.

5.2.4 Cluster-Based Journaling

Search-optimized systems organize data so that it can be read and queried as fast as possible, often causing update performance to suffer [1]. This is a difficult problem in a search-optimized file system because updates are very frequent, particularly for file systems with hundreds of millions of files. Moreover, metadata must be kept safe, requiring synchronous updates. Magellan's design complicates efficient updates in two ways. First, clusters are too large to be written to disk every time they are modified. Second, K-D trees are in-memory structures; thus, information cannot be inserted into the middle of the serialized stream on disk.

To address this issue, Magellan uses a *cluster-based journaling* technique that writes updates safely to an on disk journal and updates the in-memory cluster, but delays writing the cluster back to its primary on disk location. This technique provides three key advantages. First, updates in the journal are persistent across a crash since they can be replayed. Second, metadata updates are indexed and can be searched in real-time. Third, update operations are fast because disk writes are mostly sequential journal writes that need not wait for the cluster to be written. This approach differs from most journaling file systems that use the journal as a temporary staging area and write metadata back to its primary disk location shortly after the update is journaled [128, 147]. In Magellan, the journal is a means to recreate the in memory state in case of a crash; thus, update performance is closer to that of a log-structure file system [138].

Cluster-based journaling allows updates to achieve good disk utilization; writes are either streaming sequential writes to the journal or large sequential cluster writes. Since clusters are managed by the cluster cache, it can exploit temporal locality in workloads as was shown in Section 3.3, allowing it to keep frequently-updated clusters in memory, updating them on disk only when they become "cold". This approach also allows many metadata operations to be reflected in a single cluster optimization and write, and allows many journal entries to be freed at once, further improving efficiency.

While cluster-based journaling provides several nice features it does have trade-offs. Since metadata updates are not immediately written to their primary on-disk location, the journal can grow very large because these journal entries are not trimmed until the cluster is com-

mitted to disk. Magellan allows the journal to grow to hundreds of megabytes before requiring that clusters be flushed. Having a larger journal requires more memory resources as well as more journal replay time in the event of a crash. Since journal writes are a significant part of update performance, staging the journal in non-volatile memory such as MRAM or phase change memory could significantly boost performance if the hardware can be afforded. In addition to requiring additional memory space for the journal, delaying cluster writes to disk and performing them in the background still has the potential to impact foreground workloads. An update to a cluster that is being flushed back to disk will cause the update to be blocked until the flush is completed. Since flushing a cluster often requires both a rebalance and large disk write, this latency can be quite high. It is possible to lazily apply the update after the flush and have the update return prior to the flush completing, though will require extra data structures and processing. One of the benefits of designing Magellan for use in a parallel file system like Ceph is that bandwidth is abundant, meaning that cluster writes and journal writes can happen simultaneously without have to compete for I/O bandwidth.

5.3 Integrating Semantic File System Functionality

Thus far, our discussion of index and searchable file system design have focused on hierarchical namespaces. We highlighted the key limitations of these namespaces in Section 2.3. While our new index and file system designs enable more efficient search, which improves their effectiveness, hierarchical namespaces are not a well suited long-term solution. Instead it is often argued that search-based namespaces are a more effective method for managing billions of files [47, 148, 170]. In these namespaces, search is the primary access method and files are organized based on their attributes and search queries.

File systems that implement a search-based namespace are generally called semantic file systems. Semantic file systems are generally designed as a naming layer above a traditional file system or database [63, 69, 124] as we discussed in Section 2.5. While providing users with a better interface to storage, this design is not very scalable because it only changes how files are presented to users not how they are stored and indexed. As a result, current designs suffer

from many of the index and file system design limitations that we have already discussed and are not able to effectively scale.

In this section we outline the architecture of Copernicus, a semantic file system that uses new internal organization and indexing techniques to improve performance and scalability. The architecture that we propose aims to achieve several goals:

Flexible naming. The main drawback with current hierarchical file systems is their inability to allow flexible and semantic access to files. Files should be able to be accessed using their attributes *and* relationships. Thus, the file system must efficiently extract and infer the necessary attributes and relationships and index them in real-time.

Dynamic navigation. While search is extremely useful for retrieval, users still need a way to navigate the namespace. Navigation should be more expressive than just $parent \rightarrow child$ hierarchies, should allow dynamically changing (or virtual) directories and need not be acyclic. Relationships should be allowed between two files, rather than only directories and files.

Scalability. Large file systems are the most difficult to manage, making it critical that both search and I/O performance scale to billions of files. Effective scalability requires fine-grained control of file index structures that allow disk layouts and memory utilization to properly match workloads.

Backwards compatibility. Existing applications rely on hierarchical namespaces. It is critical that new file systems be able to support legacy applications to facilitate migration to a new paradigm.

5.3.1 Copernicus Design

Copernicus is designed as an object-based parallel file system so that it can achieve high scalability by decoupling the metadata and data paths and allowing parallel access to storage devices. However, Copernicus's techniques are applicable to a wide range of architectures. As mentioned in Section 2.2, object-based file systems consist of three main components: clients, a metadata server cluster (MDS), and a cluster of object-based storage devices (OSD). Clients perform file I/O directly with OSDs. File data is placed and located on OSDs using a pseudo-random hashing algorithm [181]. Metadata and search requests are submitted to the

MDS, which manages the namespace. Thus, most of the Copernicus design is focused on the MDS.

Copernicus achieves a scalable, semantic namespace using several new techniques as well as techniques derived from our previous Magellan and Spyglass designs. A dynamic graph-based index provides file metadata and attribute layouts that enable scalable search, as shown in Figure 5.4. Files that are semantically similar and likely to be accessed together are grouped into *clusters*, which are similar to traditional directories, and form the vertices of the graph. These clusters are similar to the cluster's Magellan used in Section 5.2, however are not grouped based on the hierarchical namespace. Inter-file relationships, such as provenance [114, 149] and temporal access patterns [155], create edges between files that enable semantic navigation. Directories are instead "virtual," and instantiated by queries. Backwards naming compatibility can be enabled by creating a hierarchical tree from the graph. Clusters store metadata and attributes in search-optimized index structures. The use of search indexes for native storage mechanisms allows Copernicus to be easily searched without additional search applications. Finally, a new journaling mechanism allows file metadata modifications to be written quickly and safely to disk while still providing real-time index updates.

Before we discuss specific design details, we present some specific examples of how Copernicus can improve how files are managed in large-scale file systems.

Understanding file dependencies. Consider a scientist running an HPC DNA sequencing application. To interpret the results, it is useful to know how the data is being generated. As the experiment runs, Copernicus allows the results to be searched in real time. If a compelling result is found, a virtual directory can be created using a query for all files from past experiments with similar results. By searching the provenance links of those files, the scientist can find which DNA sequencing libraries or input parameters are the common factor for all of the result files.

System administration. Imagine a storage administrator who discovers a serious bug in a script that has affected an unknown number of files. To locate and fix these files, the administrator can search provenance relationships to find the contaminated files (*e. g.*, files opened by the script) and build a virtual directory containing these files. A corrected version of the script can be run over the files in this directory to quickly undo the erroneous changes.

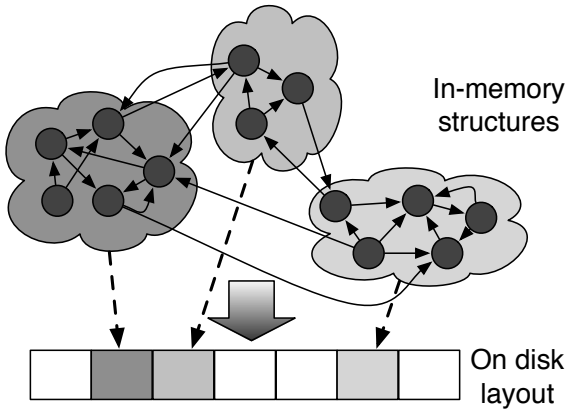


Figure 5.4: **Copernicus overview.** Clusters, shown in different colors, group semantically related files. Files within a cluster form a smaller graph based on how the files are related. These links, and the links between clusters, create the Copernicus namespace. Each cluster is relatively small and is stored in a sequential region on disk for fast access.

Finding misplaced files. Consider a user working on a paper about file system search and looking for related work. The user recalls reading an interesting paper while working on "motivation.tex" but does not know the paper's title or author. However, using temporal links and metadata, a virtual directory can be constructed of all files that were accessed at the same time as "motivation.tex", are `pdf`s, and contain "file system" and "search". The directory allows the user to easily browse the results.

5.3.2 Graph Construction

Copernicus uses a graph-based index to provide a metadata and attribute layout that can be efficiently searched. The graph is managed by the MDS. Each file is represented with an inode and is uniquely identified by its inode number. Inodes and associated attributes—content keywords and relationships—are grouped into physical clusters based on their semantic similarity. Clusters are like directories in that they represent a physical grouping of related files likely to be accessed together, in the same way that file systems try to keep files adjacent to their containing directory on disk. This grouping provides a flexible, fine-grained way to

access and control files. However, unlike directories, cluster groupings are semantic rather than hierarchical and are transparent to users: Clusters only provide physical organization for inodes. Given a file's inode number, a pseudo-random placement algorithm, CRUSH [181], identifies the locations of the file's data on the OSDs, meaning data pointers are not stored within the inode.

Inodes are grouped into clusters using *clustering policies*, which define their semantic similarity. Clustering policies may be set by users, administrators, or Copernicus, and can change over time, allowing layouts to adjust to the current access patterns. Inodes may move between clusters as their attributes change. Example clustering policies include clustering files for a common project (*e. g.*, files related to an HPC experiment), grouping files with shared attributes (*e. g.*, files owned by Andrew or all virtual machine images), or clustering files with common access patterns (*e. g.*, files often accessed in sequence or in parallel). Previous work has used Latent Semantic Indexing (LSI) as a policy to group related files [80]. In Copernicus, files are allowed to reside in only one cluster because maintaining multiple active replicas makes synchronization difficult. Clusters are kept relatively small, around 10^5 files, to ensure fast access to any one cluster; thus, a large file system may have 10^4 or more clusters.

Copernicus creates a namespace using the semantic relationships that exist between files. Relationships are directed and are represented as triples of the form $\langle$*relationship type*, *source file*, *target file*$\rangle$, and can define any kind of relationship. Relationship links may exist between files within the same or different clusters as illustrated in Figure 5.4. The graph need not be acyclic, permitting more flexible relationships. Relationship links are created implicitly by Copernicus depending on how files are used and can also be created explicitly by users and applications. Unlike a traditional file system, links only exist between two files; directories in Copernicus are "virtual" and simply represent the set of files matching a search query. These virtual directories are represented by an inode in the graph and the associated query is stored in the data region on the OSDs pointed to by the inode (similar to how directory entries are pointed to by the directory's inode in traditional file systems). A virtual directories query is periodically evaluated to ensure the results are not too stale. Users can also force the file system to execute

the query again at any time. Combining virtual directories and links provides an easy way to represent related files and navigate the namespace as well as provide backwards compatibility.

5.3.3 Cluster Indexing

Files in Copernicus may be retrieved using their metadata, content, or relationship attributes. Each cluster stores these attributes in separate search-optimized index structures, improving efficiency by allowing files to easily be searched without a separate application. File metadata is represented as $\langle attribute, value \rangle$ pairs and includes simple POSIX metadata and extended attributes. Metadata is indexed in a in-memory, multi-dimensional binary search tree called a K-D tree [24], which we also used in our Spyglass and Magellan designs. Since clusters are relatively small, each K-D tree can often be stored in a sequential region on disk. This layout, which is similar to embedded inodes [57], provides fast read access and prefetching of related metadata.

Relationship attributes are also stored in a K-D tree that has three dimensions for the three fields in the relationship triple. K-D trees allow any combination of the relationship triple to be queried. If a relationship exists between files in different clusters, the cluster storing the source file's inode indexes the relationship, to prevent duplication.

Each cluster stores full-text keywords, which are extracted from its files' contents using application-specific *transducers*, in its own inverted index. This design allows keyword search at the granularity of clusters and helps keep posting lists small so that they can be kept sequential on disk. A global indirect index, which we introduced in Section 4.2, is used to identify which clusters contain posting lists for a keyword. As mentioned previously, an indirect index consists of a keyword dictionary with each keyword entry pointing to a list of $\langle$*cluster*, *weight*$\rangle$ pairs, allowing the MDS to quickly identify the clusters most likely to contain an answer to a query and rule out those clusters that *cannot* satisfy the query.

5.3.4 Query Execution

All file accesses (*e. g.*, `open()` and `stat()`) translate to queries over the Copernicus graph index. While navigation can be done using graph traversal algorithms, queries must also be able to identify the clusters containing files relevant to the search. Since semantically related files are clustered in the namespace, it is very likely that the vast majority of clusters do not need to be searched. We showed this to be the case in Section 4.1, despite only modest semantic clustering. Additionally, Copernicus employs an LRU-based caching algorithm to ensure that queries for hot or popular clusters do not go to disk.

For file metadata and relationships, Copernicus identifies relevant clusters using *Bloom filters* [26]—bit arrays with associated hashing functions that compactly describe the contents of a cluster. Bloom filters provided similar functionality in Magellan as did signature files in Spyglass. Each cluster maintains a Bloom filter for each metadata attribute that it indexes. In addition each cluster maintains three Bloom filters that describe the relationships that it contains. These Bloom filters describe each of the fields in the $\langle$*relationship type*, *source file*, *target file*$\rangle$ relationship triple. When a cluster stores a metadata or relationship attribute, it hashes its value to a bit position in a bloom filter, which is then set to one. To determine if a cluster contains any files related to a query, the values in the query are also hashed to bit positions, which are then tested. If, and only if, all tested bits are set to one is the cluster read from disk and searched. To ensure fast access, bloom filters are kept in memory. To do this, each is kept small: 10^3 to 10^5 bit positions per bloom filter. While false positives can occur when two values hash to the same bit position, the only effect is that a cluster is searched when it does not contain files relevant to the query, degrading search performance but not impacting accuracy.

The sheer number of possible keywords occurring in file content make Bloom filters ineffective for keyword search. However, the indirect index allows fast identification of the clusters containing posting lists for the query keywords. For each keyword in the query, the list of clusters containing the keyword is retrieved. Assuming Boolean search, the lists are then intersected, producing the set of clusters that appeared in all lists. Only the posting lists from the clusters appearing in this set are retrieved and searched. The weights can be used to further

optimize query processing, first searching in clusters that are most likely to contain the desired results.

5.3.5 Managing Updates

Copernicus must effectively balance search and update performance, provide real-time index updates, and provide the data safety that users expect. Copernicus uses an approach similar to the cluster-based journaling method used in Magellan for managing metadata and relationship updates, and a client-based approach for managing content keywords. When file metadata or relationships are created, removed or modified, the update is first written safely to a journal on disk. By first journaling updates safely to disk, Copernicus is able to provide needed data safety in case of a crash. The K-D tree containing the file's inode or relationship information is then modified and marked as dirty in the cache, thereby reflecting changes in the index in real-time. When a cluster is evicted from the cache, the entire K-D tree is written sequentially to disk and its entries are removed from the journal. Copernicus allows the journal to grow up to hundreds of megabytes before it is trimmed, which helps to amortize multiple updates into a single disk write.

As mentioned in Section 5.2, K-D trees do not efficiently handle frequent inserts and modifications. Inserting new inodes into the tree can cause it to become unbalanced, degrading search performance. Again like Magellan, K-D trees are re-balanced before they are written to disk. Also, inode modifications first require the original inode to be removed and then a new inode to be inserted. Both of these operations are fast compared to writing to the journal, but since disk speed dictates update performance, storing the journal in NVRAM can significantly boost performance.

Clients write file data directly to OSDs. When a file is closed, Copernicus accesses the file's data from the OSDs and use a transducer to extract keywords. To aid this process, clients submit a list of write offsets and lengths to the MDS when they close a file. These offsets tell the MDS which parts of the file to analyze and can greatly improve performance for large files. Cluster posting lists are then updated with extracted keywords. Since cluster posting

lists are small, an in-place update method [96] can be used, ensuring that they remain sequential on disk.

5.4 Experimental Results

In this section we evaluate the performance of our searchable file system designs. To do this we evaluate a prototype implementation of Magellan. Our Copernicus design is still in its early stages and there are a number of practical design questions that must be addressed before it can be evaluated. However, since the two share a number of design features, this evaluation is relevant to Copernicus. Our current evaluation seeks to examine the following questions: (1) How does Magellan's metadata indexing impact performance? (2) How does our journaling technique affect metadata updates? (3) Does metadata clustering improve disk utilization? (4) How does our prototype's metadata performance compare to other file systems? (5) What kind of search performance is provided? Our evaluation shows that Magellan can search millions of files, often in under a second, while providing performance comparable to other file systems for a variety of workloads.

5.4.1 Implementation Details

We implemented our prototype as the metadata server (MDS) for the Ceph parallel file system [180], for several reasons. First, parallel file systems often handle metadata and data separately [34, 180]: metadata requests are handled by the MDS while data requests are handled by separate storage devices, allowing us to focus solely on MDS design. Second, Ceph targets the same large-scale, high-performance systems as Magellan. Third, data placement is done with a separate hashing function [181], freeing Magellan from the need to perform data block management. Like Ceph, our prototype is a Linux user process that uses a synchronous file in a local `ext3` file system for persistent storage.

In our prototype, each cluster has a maximum of 2,000 directories and a soft limit of 20,000 inodes, keeping them fast to access and query. We discuss the reasoning behind these numbers later in this section. K-D trees were implemented using `libkdtree++` [97], version

Attribute	Description	Attribute	Description
ino	inode number	ctime	change time
pino	parent inode number	atime	access time
name	file name	owner	file owner
type	file or directory	group	file group
size	file size	mode	file mode
mtime	modification time		

Table 5.1: **Inode attributes used.** The attributes that inodes contained in our experiments.

0.7.0. Each inode has eleven attributes that are indexed, listed in Table 5.1. Each Bloom filter is about 2 KB in size—small enough to represent many attribute values while not using significant amounts of memory. The hashing functions we use for the file size and time attributes allow bits to correspond to ranges of values. Each cluster's inode cache is around 10 KB in size, which can cache pointers to at most roughly 10% of the clusters inodes. Given the maximum cluster size, clusters generally contain between one to two MB of inode metadata. When a cluster's inode cache is full, the ratio of memory used for inode metadata to the metadata describing the cluster (*e. g.*, Bloom filters, inode caches) is roughly 40:1. However, cluster caches contain inode pointers for the most recently accessed inodes which means they generally do not use close to the full 10 KB and only contain pointers when the cluster is in memory. While our prototype implements most metadata server functionality, there are a number of features not yet implemented. Among these are hard or symbolic links, handling of client cache leases, and metadata replication. None of these functions present a significant implementation barrier, and none should significantly impact performance.

All of our experiments were performed on an Intel Pentium 4 machine with dual 2.80 GHz CPUs and 3.1 GB of RAM. The machine ran CentOS 5.3 with Linux kernel version 2.6.18. All data was stored on a directly attached Maxtor ATA 7Y250M0 7200 RPM disk.

5.4.2 Microbenchmarks

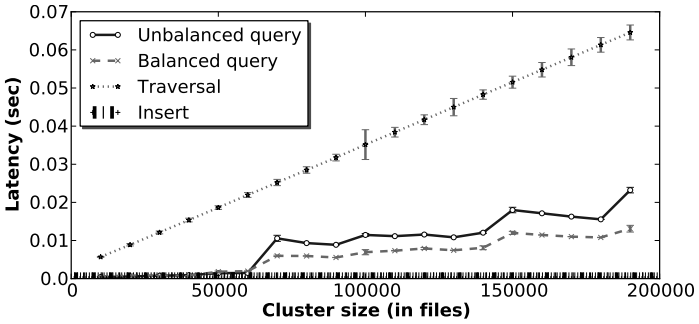
We begin by evaluating the performance of individual Magellan components using microbenchmarks.

5.4.2.1 Cluster Indexing Performance

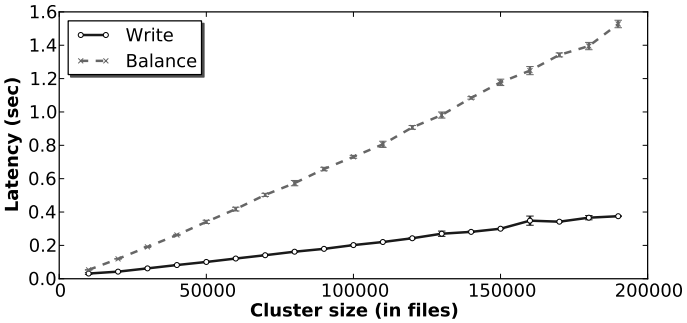
We evaluated the update and query performance for a *single* cluster in order to understand how indexing metadata in a K-D tree affects performance. Figure 5.5(a) shows the latencies for creating and querying files in a single cluster as the cluster size increases. Results are averaged over five runs with the standard deviations shown. We randomly generated files because different file systems have different attribute distributions that can make the K-D tree un-balanced and bias results in different ways [4]. The queries we used used were range queries for between two and five attributes.

We measured query latencies in a balanced and unbalanced K-D tree, as well as brute force traversal. Querying an unbalanced K-D tree is $5 - 15\times$ faster than a brute force traversal, which is already a significant speed up for just a single cluster. Unsurprisingly, brute force traversal scales linearly with cluster size; in contrast, K-D tree query performance scales mostly sub-linearly. However, it is clear that K-D tree organization impacts performance; some queries in a tree with 70,000 files are 10% slower than queries across 140,000 files. A balanced cluster provides a 33–75% query performance improvement over an unbalanced cluster. However, when storing close to 200,000 files, queries can still take longer than 10 ms. While this performance may be acceptable for "real" queries, it is too slow for many metadata look ups, such as path resolution. Below 50,000 files, however, all queries require hundreds of microseconds, assuming the cluster is already in memory.

The slow performance at large cluster sizes demonstrates the need to keep cluster sizes limited. While an exact match query in a K-D tree (*i. e.*, all indexed metadata values are known in advance) takes $O(\log N)$ time, these queries typically aren't useful because it is rarely the case that *all* metadata values are known prior to accessing a file. Instead, many queries are range queries that use fewer than k-dimensions. These queries requires $O(kN^{1-1/k})$



(a) Query and insert performance.



(b) Write and optimize performance.

Figure 5.5: **Cluster indexing performance.** Figure 5.5(a) shows the latencies for balanced and unbalanced K-D tree queries, brute force traversal, and inserts as cluster size increases. A balanced K-D tree is the fastest to search and inserts are fast even in larger clusters. Figure 5.5(b) shows latencies for K-D tree rebalancing and disk writes. Rebalancing is slower because it is requires $O(N \times \log N)$ time.

time, where N is the number of files, and k is the dimensionality, meaning that performance increasingly degrades with cluster size. While hundred microsecond latencies are acceptable, Magellan further improves performance using the hash table based inode cache that each cluster maintains for recently accessed inodes.

In contrast to query performance, insert performance remains fast as cluster size increases. The insert algorithm is similar to the exact match query algorithm, requiring only $O(\log N)$ time to complete. Even for larger K-D trees, inserts take less than 10 μs. The downside is that each insert makes the tree less balanced, degrading performance for subsequent queries until the tree is rebalanced. Thus, while inserts are fast, there is a hidden cost being paid in slower queries and having to rebalance the tree later.

Figure 5.5(b) shows latencies for writing a cluster to disk and rebalancing, the two major steps performed when a dirty cluster is written to disk. Each inode is roughly 100 bytes in size, meaning a cluster with 50,000 files is close 5 MB in size. Surprisingly, rebalancing is the more significant of the two steps, taking $3-4\times$ longer than writing the cluster to disk. The K-D tree rebalancing algorithm takes $O(N \times \log N)$ time, which accounts for this difference. However, even if we did not rebalance the K-D tree prior to flushing it to disk, K-D tree write performance is not fast enough to be done synchronously when metadata is updated as they can take tens to hundreds of milliseconds. Since a K-D tree is always written asynchronously, its performance does not affect user operation latencies, though it *can* impact server CPU utilization.

5.4.2.2 Update Performance

To evaluate how cluster-based journaling impacts update performance, we used a benchmark that creates between 100,000 and 2,000,000 files, and measured the throughput at various sizes. To do this, we used the metadata traces that we collected from three storage servers deployed at NetApp and studied in Section 3.7. We used different traces because each has different namespace organizations that impact performance [4] (*e. g.*, having few very large directories or many small directories). The servers were used by different groups within NetApp: a web server (Web), an engineering build server (Eng), and a home directory server

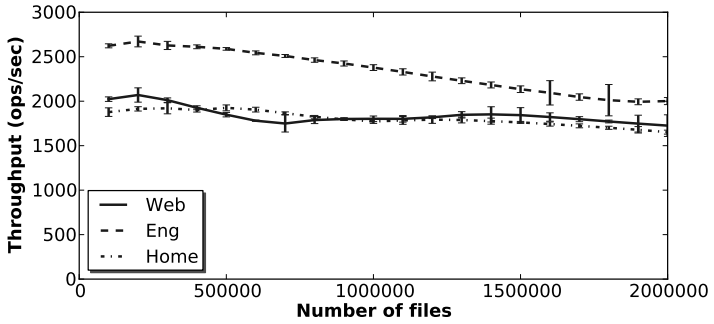


Figure 5.6: **Create performance.** The throughput (creates/second) is shown for various system sizes. Magellan's update mechanism keeps create throughput high because disk writes are mostly to the end of the journal, which yields good disk utilization. Throughput drops slightly at larger sizes because more time is spent searching clusters.

(Home). Files were inserted in the order that they were crawled; since multiple threads were used in the original crawl, the traces interleave around ten different crawls each doing depth-first search order.

Figure 5.6 shows the throughput averaged over five runs and standard deviations as the number of creates increases. We find that, in general, throughput is very high, between 1,500 and 2,500 creates per second, because of Magellan's cluster-based journaling. This throughput is higher than those recently published for comparable parallel file system metadata servers on comparable hardware; Ceph achieves around 1,000 creates per second [180] and Panasas achieves between 800 and 1,600 creates per second on various hardware setups [183]. Each create appends an update entry to the on-disk journal and then updates the in memory K-D tree. Since the K-D tree write is delayed, this cost is paid later as the benchmark streams largely sequential updates to disk.

However, create throughput drops slightly as the number of files in a cluster increases because the K-D tree itself is larger. While only a few operations experience latency increases due to waiting for a K-D tree to be flushed to disk, larger K-D trees also cause more inode cache misses, more Bloom filter false positives, and longer query latencies, thus increasing create latencies (*e. g.*,, because a file creation operation must check to see if the file already

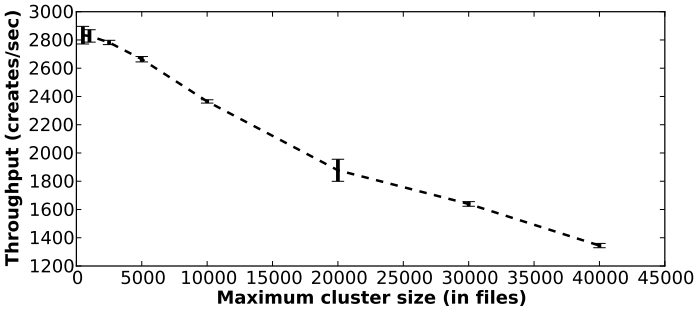
exists). In most cases, checking Bloom filters is sufficient for determining is a file already exists; again, though, the higher rate of false positives causes more K-D tree searches. Since inode caches only contain the most recently accessed files, it will not prevent K-D tree queries when Bloom filters yield false positives. There are a limited number of instances where a create operation is waiting for a cluster to be written back to disk because the create benchmark usually updates clusters in a linear fashion and only accesses other clusters for path resolution. Thus, it is usually unlike that a cluster will be accessed while it is being flushed.

5.4.2.3 Metadata Clustering

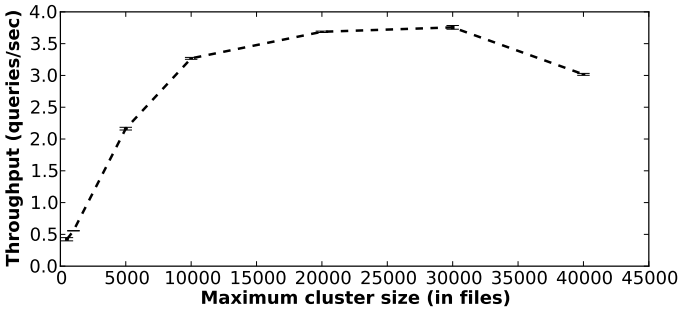
We next examined how different maximum cluster sizes affects performance and disk utilization. To do this, we evaluated Magellan's create and query throughputs as its maximum cluster size increases. The maximum cluster size is the size at which Magellan tries to cap clusters. If a file is inserted, it is placed in the cluster of its parent directory, regardless of size. For a directory, however, Magellan creates a new cluster if the cluster has too many directories or total inodes. Maximum cluster size refers to the maximum inode limit; we set the maximum directory limit to $1/10^{th}$ of that.

Figure 5.7(a) shows the total throughput for creating 500,000 files from the Web trace over five runs as the maximum cluster size varies from 500 to 40,000 inodes. As the figure shows, create throughput steadily decreases as maximum cluster size increases. While the throughput at cluster size 500 is around 2,800 creates per second, at cluster size 40,000, which is an $80\times$ increase, throughput drops roughly 50%. Disk utilization is not the issue, since both use mostly sequential disk writes. Rather, the decrease is primarily due to having to operate on larger K-D trees. Smaller clusters have more effective inode caching (less data to cache per K-D tree) and Bloom filters (fewer files yielding fewer false positives). Additionally, queries on smaller K-D trees are faster. Since journal writes and K-D tree insert performance do not improve with cluster size, a larger maximum cluster size has little positive impact.

Figure 5.7(b) shows that query performance scales quite differently from create performance. We used a simple query that represented a user searching for a file she owns with a particular name (*e. g.*, filename equal to `mypaper.pdf` and owner id equal to `3704`). We



(a) Create latencies



(b) Query latencies.

Figure 5.7: **Metadata clustering.** Figure 5.7(a) shows create throughput as maximum cluster size increases. Performance decreases with cluster size because inode caching and Bloom filters become less effective and K-D tree operations become slower. Figure 5.7(b) shows that query performance is worse for small and large sizes.

Name	Application	Metadata Operations
Multiphysics A	Shock physics	70,000
Multiphysics B	Shock physics	150,000
Hydrocode	Wave analysis	210,000
Postmark	E-Mail and Internet	250,000

Table 5.2: **Metadata workload details.**

find that query throughput *increases* $7-8\times$ as maximum cluster size varies from 500 to 25,000. When clusters are small, metadata clustering is not as helpful because many disk seeks may still be required to read the needed metadata. As clusters get larger disk utilization improves. However, throughput decreases 15% when maximum cluster size increases from 30,000 to 40,000 files. When clusters are too large, time is wasted reading unneeded metadata, which can also displace useful information in the cluster cache. In addition, larger K-D trees are slower to query. The "sweet spot" seems to be around 20,000 files per cluster, which we use as our prototype's default and which works well for our experiments. While the precise location of this "sweet spot" will vary between workloads, the general trend that we observe will be consistent across workloads.

5.4.3 Macrobenchmarks

We next evaluated general file system and search performance using a series of macrobenchmarks.

5.4.3.1 File System Workload Performance

We compared our prototype to the original Ceph MDS using four different application workloads. Three workloads are HPC application traces from Sandia National Laboratory [143] and the other is the Postmark [84] benchmark. Table 5.2 provides additional workload details. We used HPC workloads because they represent performance critical applications. Postmark was chosen because it presents a more general workload, and is a commonly used benchmark.

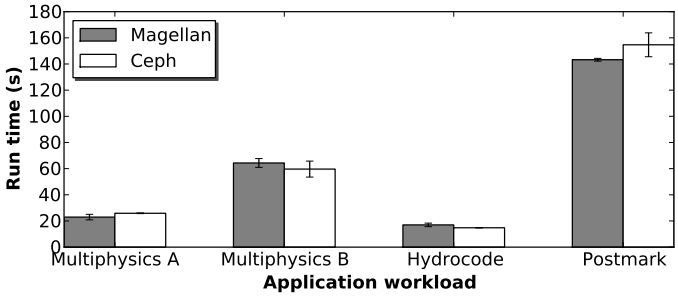


Figure 5.8: **Metadata workload performance comparison.** Magellan is compared to the Ceph metadata server using four different metadata workloads. In all cases, both provide comparable performance. Performance differences are often due to K-D tree utilization.

	Multiphysics A	Multiphysics B	Hydrocode	Postmark
Magellan	3041	2455	10345	1729
Ceph	2678	2656	14187	1688

Table 5.3: **Metadata throughput (ops/sec).** The Magellan and Ceph throughput for the four workloads.

While the benchmarks are not large enough to evaluate all aspects of file system performance (many common metadata benchmarks are not [174]), they are able to highlight some important performance differences. We modified the HPC workloads to ensure that directories were created before they were used. We used Postmark version 1.51 and configured it to use 50,000 files, 20,000 directories, and 10,000 transactions. All experiments were performed with cold caches.

Figure 5.8 shows the run times and standard errors averaged over five runs, with the corresponding throughputs shown in Table 5.3. Total run times are comparable for both, showing that Magellan is able to achieve similar file system performance to the original Ceph MDS. However, performance varies between the two; at most, our prototype ranges from 13% slower than Ceph to 12% faster. As in the previous experiments, a key reason for performance decreases was K-D tree performance. The Multiphysics A and Multiphysics B traces have very similar distributions of operations, though Multiphysics B creates about twice as many

Set	Query	Metadata Attributes
Set 1	Where is this file located?	Retrieve files using filename and owner.
Set 2	Where, in this part of the system, is this file located?	Use query 1 with an additional directory path.
Set 3	Which of my files were modified near this time and at least this size?	Retrieve files using owner, mtime, and size.

Table 5.4: **Query Sets.** A summary of the searches used to generate our evaluation query sets.

files. Table 5.3 shows that between the two, our prototype's throughput drops by about 20% from about 3,000 operations per second to 2,500, while Ceph's throughput remains close to consistent. This 20% overhead is spent almost entirely doing K-D tree searches.

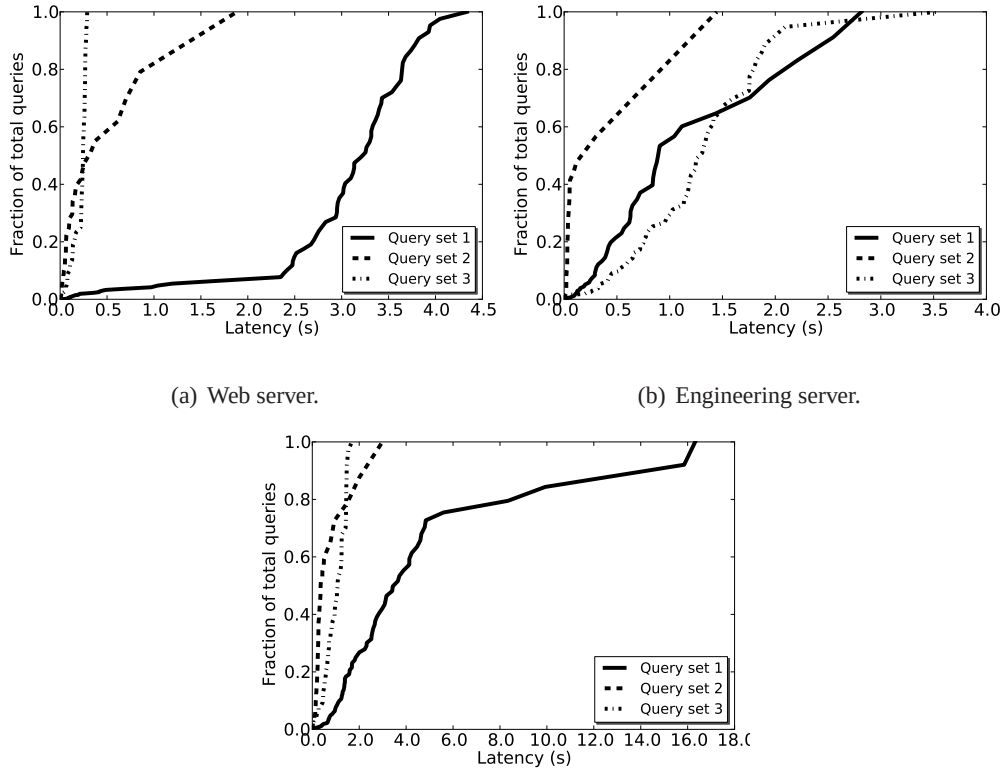
Our prototype yields a 12% performance improvement over Ceph for the Postmark workload. Postmark creates a number of files consecutively which benefit from cluster-based journaling. Throughput for these operations can be up $1.5 - 2\times$ faster than Ceph. Additionally, the ordered nature of the workload produces good inode cache hit ratios (path resolution look ups frequently hits the inode cache because these inodes were recently created).

These workloads show differences and limitations (*e. g.*, large K-D tree performance) with our design, though they indicate that it can achieve good file system performance. A key reason for this is that, while Magellan makes a number of design changes, it keeps the basic metadata structure (*e. g.*, using directory and file inodes, organizing inodes into a physical hierarchy). This validates an important goal of our design: Address issues with search performance while maintaining many aspects that current metadata designs do well.

5.4.3.2 Search Performance

To evaluate search performance, we created three file system images using the Web, Eng, and Home metadata traces with two, four, and eight million files, respectively. The cluster cache size is set to 20, 40, and 80 MB for each image, respectively, so that searches are not performed solely in memory. Before running the queries, we warmed the cluster cache with random cluster data.

Unfortunately, there are no standard file system search benchmarks. Instead, we generated synthetic query sets based on queries that we believe represent common metadata search



(a) Web server.

(b) Engineering server.

(c) Home server.

Figure 5.9: **Query execution times.** A CDF of query latencies for our three query sets. In most cases, query latency is less than a second even as system size increases. Query set 2 performs better than query set 1 because it includes a directory path from where Magellan begins the search, which rules out files not in that sub-tree from the search space.

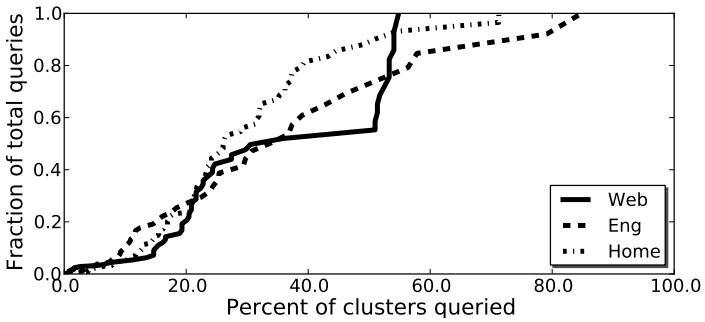


Figure 5.10: **Fraction of clusters queried.** A CDF of the fraction of clusters queried for query set 1 on our three data sets is shown. Magellan is able to leverage Bloom filters to exploit namespace locality and eliminate many clusters from the search space.

use cases. The queries and the attributes used are given in Table 5.4. Query attributes are populated with random data from the traces, which allows the queries to follow the same attribute distributions in the data sets while providing some randomness. Query set 1 and 2 produced very few search results (generally around one to five) while query set 3 could yield thousands of results.

Figure 5.9 shows the cumulative distribution functions (CDF) for our query sets run over the three different traces. Our prototype is able to achieve search latencies that are less than a second in most cases, even as file system size increases. In fact, all queries across all traces are less than six seconds, with the exception of several in the Home trace that were between eight and fifteen seconds. Evaluating query set 1 using a brute force search, which is the only search option if no separate search application is available, took 20 minutes for the Web trace and 80 minutes on the Home trace on a local `ext3` file system on the same hardware configurations. Compared to Magellan, brute force search is up to 4–5 orders of magnitude slower. In addition, to the search-optimized on disk layout and inode indexing, Magellan is able to leverage namespace locality by using Bloom filters to eliminate a large fraction of the clusters from the search space. Figure 5.10 shows a CDF of the fraction of clusters accessed for a query using query set 1 on all three of our traces. We see that 50% of queries access fewer than 40% of the clusters in all traces. Additionally, over 80% of queries access fewer than 60% of the

clusters. Query set 2 includes a directory path from which to begin the search, which explicitly limits the search space. As a result, search performance for query set 2 is consistently fastest. However, in some cases query set 1 is faster than query set 2. This is because the queries for query set 2 use a different set of randomly selected attribute values than query set 1, which can alter run time and cache hit ratios.

While queries typically run in under a second, some queries take longer. For example, latencies are mostly between 2 and 4 seconds for query set 1 on our Web data set in Figure 5.9(a). In these cases, many of the clusters are accessed from disk, which increases latency. The Web trace contained a lot of common file names (*e. g.*, `index.html` and `banner.jpg`) that were spread across the file system. We believe these experiments show that Magellan is capable of providing search performance that is fast enough to allow metadata search to be a primary way for users and administrators to access and manage their files.

5.5 Summary

The rapid growth in data volume is changing how we access and manage our files. Large-scale systems increasingly require search to better locate and utilize data. While search applications that are separate from the file system are adequate for small-scale systems, they have inherent limitations when used as large-scale, long-term file search solutions. We believe that a better approach is to build search functionality directly into the file system itself.

In this chapter, we analyzed the hypothesis that it is possible to enable effective search performance directly within the file system without sacrificing file system performance. We presented the design of a new file system metadata architecture called Magellan that enables metadata to be efficiently searched while maintaining good file system performance. Unlike previous solutions that relied on relational databases, Magellan uses several novel search-optimized metadata layout, indexing, and update techniques. Additionally, we outlined the architecture of Copernicus, as scalable, semantic file system. Unlike previous semantic file system designs which are designed as naming layers on top of a normal file system or database, Copernicus uses a novel graph-based index to enable a scalable search-based namespace and namespace naviga-

tion using inter-file relationships. Using real-world data, our evaluation showed that Magellan can search over file systems with millions of files in less than a second and provide file system performance comparable to other systems. While Magellan and Copernicus's search-optimized designs do have limitations, they demonstrates that search and file systems can be effectively combined, representing a key stepping stone in the path to enabling better ways to locate and manage our data.

Chapter 6

Future Directions

In this thesis we present a very different approach to organizing and indexing files in large-scale file systems. While we discuss some initial designs and findings that support our hypotheses, this research area is in its nascent stages and there are number of practical trade-offs and design issues that need to be addressed. In addition, there a number of ways in which this work can be extended in the future to further improve data management in large-scale file systems. We outline possible future work for each chapter individually.

6.1 Large-Scale File Systems Properties

There are a number of studies that can impact file system search designs that have not been examined before. Our examination of file system workloads looked at many traditional properties such as access sequentially and access patterns. Many of our initial observations can be extended to provide additional useful information. For example, we found that a small fraction of clients can constitute a large portion of the workload traffic. Taking a closer look at the access patterns of individual clients, particularly the "power clients" who generate significant traffic, could provide valuable insights into how the workload properties are distributed across the clients and how file systems can better tailor data organization, caching, and prefetching for client needs. Also, our study found that metadata requests constitute roughly half of all requests, making a closer examination of metadata workloads important. In particular, the tem-

poral correlations between metadata requests (*e. g.*, in Linux `readdir()` may be commonly followed by multiple `stat()` operations) are highly relevant.

An important area that has not been thoroughly examined is spatial namespace locality. Thus far, we have examined only how the namespace impacts metadata attribute values. It is important to also know how workload properties differ for various parts of the namespace. For example, directories containing read-only system files may have very different workload properties than a directory storing database or log files. Understanding how namespace location impacts workload can be used to improve disk layout, prefetching, and caching designs. Previous work has demonstrated potential file system performance benefits using file type and size attribute information to guide on disk layout [190], which we have shown are influenced by location in the namespace. Our index designs and other previous file systems [57, 182] assume some amount of namespace locality in the workload but a proper characterization, such as calculating the distribution of requests across directories in the namespace, is important. Additionally, understanding namespace locality can improve how file systems organize and index data for different parts of the namespace.

Our Spyglass metadata index design benefited from empirical analysis of real-world metadata properties. Similarly, large-scale web search engine designs [15] have benefited from studies of web page content and keyword properties [19, 45]. However, to the best of our knowledge, no equivalent study has been conducted for large-scale file system content keywords. The collection and analysis of file keywords will help guide future file system inverted index designs. Particular areas of interest include keyword namespace locality and keyword correlations for ranking. An important consideration for keyword collection in file systems is security as many users will not want their data analyzed in the clear.

6.2 New Approaches to File Indexing

Our index designs are among the first meant specifically for file systems. However, there is only limited knowledge of how file system users will use search and only a limited corpus of available file search data from which to draw design decisions. As a result, there

are many design trade-offs and improvements that can still be made as search becomes more ubiquitous. In our designs we only leveraged namespace locality to partition the index. However, using other partitioning mechanisms may provide additional benefits. Partitioning based on access patterns may be able to improve cost-effectiveness. For example, files that have not been accessed in over six months may be rarely queried since in most cases querying for a file is indicative of a future access. These files can be grouped together and possibly migrated to cheaper, lower-tier storage. Other mechanisms, such as, machine learning clustering algorithms, classification algorithms, or provenance may also be useful.

Another important area that we have not fully addressed is how to distribute the index across a large-scale file system. It is likely that in large-scale file systems the index will have to be distributed to achieve the needed performance and scalability. However, there are a number of trade-offs that must be considered when distributing the index and many of these depend on the file system's architecture. For example, in a clustered NAS environment [46], it may be appropriate to build an inverted index on each file server since each manages an entire sub-tree (*e. g.*, a logical volume) and files are not striped across servers. However, in a parallel file system [62], it may be beneficial to centralize some parts of the index (*e. g.*, the dictionary) at the metadata server for management purposes. Additionally, co-locating index structures with the files they index can greatly improve update performance.

6.3 Towards Searchable File Systems

We presented the designs of two file systems that use new internal organizations that allow files to be efficiently searched. These file systems are very different from traditional designs and thus a number of practical questions remain. The use of K-D trees in Magellan and Copernicus provided a reasonably effective method for multi-dimensional attribute search. However, it does have drawbacks, such as being an in-memory only data structure, needing to be rebalanced for better performance, and providing poor performance when large. Thus, looking at how other multi-dimensional index structures can improve or enable new functionality is important. For example, FastBit [187] provides high compression ratios that trade-off CPU

with disk and cache utilization. Also, K-D-B-trees [136] are K-D trees that need to only reside partially in-memory. While this structure may negate some of the prefetching of metadata clustering, it may reduce unneeded data being read or written to disk.

Thus far our work has not looked at how to enforce file security permissions in search. Doing so is an important problem because many data sets, such as medical records and scientific results, are in need of effective search but have highly sensitive data. As discussed in Section 2.4, enforcing security while maintaining good performance is difficult. Existing solutions either ignore permissions [66], build a separate index for each user [110] that adds significant space and update overhead, or perform permission checks for every search result (*e. g.,* `stat()`) that degrades search performance and pollutes the file cache. Additionally, permission changes must be synchronously applied to the index, otherwise a security leak is possible. One approach to this problem is to embedded security permission information into the metadata clustering. This approach can partition the index along permission boundaries and use this information to eliminate clusters from the search space that the user does not have permission to access. Doing so can help reduce the size of the search space while enforcing file permissions.

Additionally, the dynamic graph-based index that Copernicus uses still has a number of basic questions that need to be resolved. For example, it is expected that a general graph that is based on inter-file relationships will perform similar to current hierarchical graphs. However, it is unclear if it can provide the kinds of efficient update performance under normal to intense file system workloads. An effective implementation is needed to verify theses ideas and understand the differences. Additionally, automatic ways to extract inter-file relationships and proper clustering attributes are needed.

Chapter 7

Conclusions

The world is moving towards a digital infrastructure, creating an unprecedented need for data storage. Today's file systems must store petabytes of data across billions of files and may be storing exabytes of data and trillions of files in the near future [58]. This data storage need has introduced a new challenge: How do we effectively manage and organize such large file systems? This is a challenge because large-scale file systems organize files using a hierarchical namespace that was designed over forty years ago for file systems containing less than 10 MB [38]. This organization is restrictive, difficult to use, and can limit scalability. As a result, there has been increasing demand for search-based file access, which allows users to access files by describing *what* they want rather than *where* it is.

Unfortunately, large-scale file systems are difficult to search. Current file system search solutions are designed as applications that are separate from the file system and utilize general-purpose index structures to provide search functionality. Keeping search separate from the file system leads to consistency and efficiency issues at large-scales and general-purpose indexes are not optimized for file system search, which can limit their performance and scalability. As a result, current solutions are too expensive, slow, and cumbersome to be effective at large-scales.

This thesis has demonstrated several novel approaches to how files are organized, indexed, and searched in large-scale file systems. We hypothesized that more effective search can be achieved using new index structures that are specifically designed for file systems and

new file system designs that better integrate search-based access. We explored these hypotheses we three main contributions: (1) a fresh look at large-scale file system properties using workload and snapshot traces, (2) new index designs for file metadata and content that leverage these properties, (3) and new file metadata and semantic file system designs that directly integrate search functionality.

We now describe the conclusions of each specific contribution.

Properties of large-scale file systems: We measured and analyzed file system workload and snapshot traces collected from several large-scale file systems in the NetApp data center. Our study represents the first major workload study since 2001 [48], the first large-scale analysis of CIFS [92] workloads, and the first to study large-scale enterprise snapshot and workload traces in over a decade.

Our analysis showed that a number of important file system workload properties, such as access patterns and sequentially, have changed since previous studies and are different on network file systems than on previous local file systems. Additionally, we found new observations regarding file sharing, file re-access, metadata attribute distributions, among others. Some of our important findings include workloads are more write-heavy than in the past: read to write byte ratios are only 2:1, compared to 4:1 or higher in past studies. Also, a large portion of file data is cold with less than 10% of total storage being accessed during our three month tracing period and files are infrequently re-accessed with 66% of opened files not being accessed again. We found that metadata attribute values are heavily clustered in the namespace. Attribute values we studied occurred in fewer than 1% of the total directories. Also, metadata attribute distribution are highly skewed though their intersections are more uniformly distributed. We discussed how these observation can impact future file system design and organization.

New approaches to file indexing: We developed two new index structures that aim to improve search and update performance and scalability by leveraging the file system properties that we observed. We presented designs of an index for file metadata and an index for file content search. Unlike general-purpose indexes that current solutions rely on, our

index designs are the first that are meant specifically for file systems and which exploit file system properties. We introduced hierarchical partitioning as a method for providing flexible index control that leveraged namespace locality. Our metadata index used signature files to reduce the search space during query execution and our content index used a new index structure called the indirect index for the same purpose. Our metadata index introduced partition-based versioning to provide fast update performance, while our content index used a merge-based algorithm to update posting lists in our indirect index. An evaluation of our metadata index using real-world trace data showed that it can provide search performance that is up to 1–4 orders of magnitude faster than basic DBMS setups, while providing update performance that is up to $40\times$ faster and using less than 0.1% of the file system's disk space. These results showed that file system search performance and scalability can be significantly improved with specialized index designs.

Towards searchable file systems: We designed two new file systems that can directly provide file search. Rather than rely on external search applications which face significant consistency and efficiency problems at large-scales, our designs use novel internal data layout, indexing, and update algorithms to provide fast file search directly within the file system. We introduced a new metadata architecture, called Magellan, that uses a novel metadata layout to improve disk utilization for searches. Also, inodes are stored in multi-dimensional index structures that provide efficient multi-attribute search and a new journaling mechanism allows fast and reliable metadata updates. We also outlined the design of a new semantic file system called Copernicus. Unlike previous semantic file system designs that used a basic naming layer on top of a traditional file system or database, Copernicus uses a novel graph-based index to provide a dynamic, searchable namespace. Semantically related files are clustered together and inter-file relationships allow navigation of the namespace. An evaluation of our Magellan prototype showed that searches could often be performed in under a second. Additionally, performance for normal file system workloads ranged from 13% slower to 12% faster than the Ceph file system. We also found that cluster-based journaling enabled good performance for

metadata updates and showed the performance trade-offs associated with indexing inodes in multi-dimensional data structures. Our evaluation demonstrated that efficient search performance can be enabled within the file system while providing normal workload performance comparable to that of other file systems.

In summary, effective data access is a primary goal of a file system, though is becoming increasingly difficult with the rapid growth of data volumes. The new file indexing and file system search designs presented in this thesis allow data to be more effectively accessed and managed at large-scales. This work, along with the new research areas that follow, should play a key role in enabling the continued construction of highly scalable file systems.

Bibliography

[1] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, Vancouver, Canada, June 2008.

[2] Atul Adya, William J. Bolosky, Miguel Castro, Ronnie Chaiken, Gerald Cermak, John R. Douceur, John Howell, Jacob R. Lorch, Marvin Theimer, and Roger 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)*, Boston, MA, December 2002. USENIX.

[3] Avnish Aggarwal and Karl Auerbach. Protocol standard for a netbios service on a tcp/udp transport. IETF Network Working Group RFC 1001, March 1987.

[4] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Generating realistic *impressions* for file-system benchmarking. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*, pages 125–138, February 2009.

[5] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A five-year study of file-system metadata. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, pages 31–45, February 2007.

[6] Sasha Ames, Nikhil Bobb, Kevin M. Greenan, Owen S. Hofmann, Mark W. Storer, Carlos Maltzahn, Ethan L. Miller, and Scott A. Brandt. LiFS: An attribute-rich file system for storage class memories. In *Proceedings of the 23rd IEEE / 14th NASA Goddard*

Conference on Mass Storage Systems and Technologies, College Park, MD, May 2006. IEEE.

[7] Sasha Ames, Carlos Maltzahn, and Ethan L. Miller. Quasar: A scalable naming language for very large file collections. Technical Report UCSC-SSRC-08-04, University of California, Santa Cruz, October 2008.

[8] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 1–14, Big Sky, MT, October 2009.

[9] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 1–14, 2009.

[10] Eric Andersen. Capture, conversion, and analysis of an intense nfs workload. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*, pages 139–152, February 2009.

[11] Apache. Welcome to Lucene! `http://lucene.apache.org/`, 2009.

[12] Apple. Spotlight Server: Stop searching, start finding. `http://www.apple.com/server/macosx/features/spotlight/`, 2008.

[13] Apple. HFS Plus volume format. `http://developer.apple.com/mac/library/technotes/tn/tn1150.html`, 2009.

[14] Apple. Spotlight Server: Stop searching, start finding. `http://www.apple.com/server/macosx/features/spotlight/`, 2009.

[15] Arvind Arasu, Junghoo Cho, Hector Garcia-Molina, Andreas Paepcke, and Sriram Raghavan. Searching the web. *ACM Transactions on Internet Technology*, 1(1):2–43, 2001.

[16] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System r: relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, 1976.

[17] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 198–212, October 1991.

[18] Beagle. Beagle – quickly find the stuff you care about. `http://beagle-project.org/`, 2009.

[19] Steven M. Beitzel, Eric C. Jensen, Abdur Chowdhury, David Grossman, and Ophir Frieder. Hourly analysis of a very large topical categorized web query log. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '04)*, Sheffield, UK, June 2004.

[20] Timothy C. Bell, Alistair Moffat, Craig G. Nevill-Manning, Ian H. Witten, and Justin Zobel. Data compression in full-text retrieval systems. *Journal of the American Soiety for Information Sciences*, 44(9):508–531, 1993.

[21] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming B-trees. In *Proceedings of the 19th Symposium on Parallel Algorithms and Architectures (SPAA '07)*, pages 81–92, 2007.

[22] J. Michael Bennett, Michael A. Bauer, and David Kinchlea. Characteristics of files in nfs environments. In *Proceedings of the 1991 ACM Symposium on Small Systems*, pages 33–40, 1991.

[23] John Bent. On the managing large-scale scientific files, 2008.

[24] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[25] Deepavali Bhagwat and Neoklis Plyzotis. Searching a file system using inferred semantic links. In *HYPERTEXT '05: Proceedings of the sixteenth ACM conference on Hypertext and hypermedia*, pages 85–87, 2005.

[26] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[27] Scott Brandt, Carlos Maltzahn, Neoklis Polyzotis, and Wang-Chiew Tan. Fusing data management services with file systems. In *Proceedings of the 4th ACM/IEEE Petascale Data Storage Workshop (PDSW '09)*, November 2009.

[28] Eric Brewer. *Readings in Database Systems*, chapter Combining Systems and Databases: A Search Engine Retrospective. MIT Press, 4th edition, 2005.

[29] Sergey Brin and Larry Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.

[30] Nader H. Bshouty and Geoffery T. Falk. Compression of dictionaries via extensions to front coding. In *Proceedings of the Fourth International Conference on Computing and Information (ICCI '92)*, pages 361–364, Washington D.C., 1992.

[31] Stefan Büttcher. *Multi-User File System Search*. PhD thesis, University of Waterloo, 2007.

[32] Stefan Büttcher and Charles L. A. Clarke. Indexing time vs. query-time trade-offs in dynamic information retrieval systems. In *Proceedings of the 14th ACM Conference on Information and Knowledge Management (CIKM 2005)*, pages 317–318, November 2005.

[33] Stefan Büttcher and Charles L. A. Clarke. A security model for full-text file system search in multi-user environments. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*, pages 169–182, San Francisco, CA, December 2005.

[34] Philip H. Carns, Walter B. Ligon, Robert B. Ross, and Rajeev Thakur. PVFS: a parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000.

[35] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET workshop on Data description, access and control (FIDET '74)*, pages 249–264, 1974.

[36] 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 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 205–218, Seattle, WA, November 2006.

[37] Edgar Frank Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 26(1):64–69, 1983.

[38] R.C. Daley and P.G. Neumann. A general-purpose file system for secondary storage. In *Proceedings of the Fall Joint Computer Conference, Part I*, pages 213–229, 1965.

[39] Shobhit Dayal. Characterizing hec storage systems at rest. Technical Report CMU-PDL-08-109, Carnegie Mellon University, Pittsburg, PA, 2008.

[40] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, December 2004.

[41] Giuseppe DeCandia, Deniz Hastorun, Gunavardhan Kakulapati Madan Jampani, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 205–220, October 2007.

[42] Cory Doctorow. Welcome to the petacentre. *Nature*, 455:16–21, September 2008.

[43] John R. Douceur and William J. Bolosky. A large-scale study of file system contents. In *Proceedings of the 1999 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Atlanta, GA, May 1999.

[44] 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)*, pages 321–334, Seattle, WA, November 2006. Usenix.

[45] Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey Mogul. Rate of change and other metrics: a live study of the world wide web. Monterey, CA, 1997.

[46] Michael Eisler, Peter Corbett, Michael Kazar, and Daniel S. Nydick. Data ONTAP GX: A scalable storage cluster. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, pages 139–152, San Jose, CA, February 2007.

[47] Daniel Ellard. The file system interface in an anachronism. Technical Report TR-15-03, Harvard University, November 2003.

[48] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive nfs tracing of email and research workloads. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)*, pages 203–216, San Francisco, CA, March 2003. USENIX.

[49] Daniel Ellard, Jonathan Ledlie, and Margo Seltzer. The utility of file names. Technical Report TR-05-03, Harvard University, March 2003.

[50] Daniel Ellard, Michael Mesnier, Eno Thereska, Gregory R. Ganger, and Margo Seltzer. Attribute-based prediction of file properties. Technical Report TR-14-03, Harvard University, Cambridge, MA, 2004.

[51] Enterprise Stragegy Groups. ESG Research Report: storage resource management on the launch pad. Technical Report ETSG-1809930, Enterprise Stragegy Groups, 2007.

[52] Facebook, Inc. Needle in a haystack: efficient storage of billions of photos. `http://www.facebook.com/note.php?note_id=76191543919`, 2009.

[53] Chris Faloutsos and Stavros Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Information Systems*, 2(4):267–288, 1984.

[54] Christos Faloutsos and Raphael Chan. Fast text access methods for optical and large magnetic disks: Designs and performance comparison. In *Proceedings of the 14th Conference on Very Large Databases (VLDB)*, Los Angeles, CA, August 1988.

[55] Fast, A Microsoft Subsidiary. FAST – enterprise search. `http://www.fastsearch.com/`, 2008.

[56] Clark D. French. One size fits all database architectures do not work for dss. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 449–450, San Jose, CA, May 1995.

[57] Gregory R. Ganger and M. Frans Kaashoek. Embedded inodes and explicit groupings: Exploiting disk bandwidth for small files. In *Proceedings of the 1997 USENIX Annual Technical Conference*, pages 1–17. USENIX Association, January 1997.

[58] John F. Gantz, Christopher Chute, Alex Manfrediz, Stephen Minton, David Reinsel, Wolfgang Schlichting, and Anna Toncheva. The diverse and exploding digital universe: An updated forecast of worldwide information growth through 2011. IDC white paper, sponsored by EMC, March 2008.

[59] Narain Gehani, H.V. Jagadish, and William D. Roome. Odefs: A file system interface to an object-oriented database. In *Proceedings of the 20th Conference on Very Large Databases (VLDB)*, pages 249–260, Santiago de Chile, Chile, 1984.

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

[61] Dominic Giampalo. *Practical File System Design with the Be File System*. Morgan Kaufmann, 1st edition, 1999.

[62] 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. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 92–103, San Jose, CA, October 1998.

[63] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole, Jr. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 16–25. ACM, October 1991.

[64] Corrado Gini. Measurement of inequality and incomes. *The Economic Journal*, 31:124–126, 1921.

[65] Goebel Group Inc. Compare search appliance tools. `http://www.goebelgroup.com/sam.htm`, 2008.

[66] Google, Inc. Google Desktop: Information when you want it, right on your desktop. `http://www.desktop.google.com/`, 2007.

[67] Google, Inc. Google enterprise. `http://www.google.com/enterprise/`, 2008.

[68] Google, Inc. We knew the web was big... `http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html`, 2008.

[69] Burra Gopal and Udi Manber. Integrating content-based access mechanisms with hierarchical file systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–278, February 1999.

[70] Jim Gray, David T. Liu, Maria Nieto-Santisteban, Alex Szalay, David J. DeWitt, and Gerd Heber. Scientific data management in the coming decade. *SIGMOD Record*, 34(4):34–41, December 2005.

[71] Steven Gribble, Gurmeet Singh Manku, Eric Brewer, Timothy J. Gibson, and Ethan L. Miller. Self-similarity in file systems: Measurement and applications. In *Proceedings*

of the 1998 SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 141–150, Madison, WI, June 1998.

[72] Gary Grider, James Nunez, John Bent, Rob Ross, Lee Ward, Steve Poole, Evan Felix, Ellen Salmon, and Marti Bancroft. Coordinating government funding of file system and I/O research through the High End Computing University Research Activity. *Operating Systems Review*, 43(1):2–7, January 2009.

[73] Karl Gyllstrom and Craig Soules. Seeing is retrieving: Building information context from what the user sees. In *IUI '08: Proceedings of the 13th International Conference on Intelligent User Interfaces*, pages 189–198, 2008.

[74] Donna Harman, R. Baeza-Yates, Edward Fox, and W. Lee. Inverted files. *Information retrieval: Data structures and algorithms*, pages 28–43, 1992.

[75] Stavros Haziopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 981–992, Vancouver, BC, Canada, 2008.

[76] Stavros Haziopoulos, Velen Liang, Daniel J. Abadi, and Samuel Madden. Performance tradeoffs in read-optimized databases. In *Proceedings of the 32nd Conference on Very Large Databases (VLDB)*, pages 487–498, Seoul, Korea, September 2006.

[77] Dave Hitz, James Lau, and Michael Malcom. File system design for an NFS file server appliance. In *Proceedings of the Winter 1994 USENIX Technical Conference*, pages 235–246, San Francisco, CA, January 1994.

[78] Allison L. Holloway and David J. DeWitt. Read-optimized databases, in depth. In *Proceedings of the 34th Conference on Very Large Databases (VLDB '08)*, pages 502–513, Auckland, New Zealand, August 2008.

[79] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. Wes. Scale and performance in a

distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[80] Yu Hua, Hong Jiang, Yifeng Zhu, Dan Feng, and Lei Tian. SmartStore: A new metadata organization paradigm with metadata semantic-awareness for next-generation file systems. In *Proceedings of SC09*, Portland, OR, November 2009.

[81] Larry Huston, Rahul Sukthankar, Rajiv Wickremesinghe, M. Satyanarayanan, Gregory R. Ganger, Erik Riedel, and Anastassia Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST)*, pages 73–86, San Francisco, CA, April 2004. USENIX.

[82] Index Engines. Power over information. `http://www.indexengines.com/online_data.htm`, 2008.

[83] Glen Jeh and Jennifer Widom. Scaling personalized web search. In *Proceedings of the 12th International World Wide Web Conference*, Budapest, Hungry, 2003.

[84] Jeffrey Katcher. PostMark: a new file system benchmark. Technical Report TR-3022, NetApp, Sunnyvale, CA, 1997.

[85] Kazeon. Kazeon: Search the enterprise. `http://www.kazeon.com/`, 2008.

[86] Kernel.org. inotify official readme. `http://www.kernel.org/pub/linux/kernel/people/rml/inotify/README`, 2008.

[87] Setrag Khoshafian, George Copeland, Thomas Jagodits, Haran Boral, and Patrick Valduriez. A query processing strategy for the decomposed storage model. In *Proceedings of the 3rd International Conference on Data Engineering (ICDE '87)*, pages 636–643, 1987.

[88] James J. Kistler and Mahadev Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.

[89] Jonathan Koren, Yi Zhang, Sasha Ames, Andrew Leung, Carlos Maltzahn, and Ethan L. Miller. Searching and navigating petabyte scale file systems based on facets. In *Proceedings of the 2007 ACM Petascale Data Storage Workshop (PDSW '07)*, pages 21–25, Reno, NV, November 2007.

[90] Jonathan Koren, Yi Zhang, and Xue Liu. Personalized interactive faceted search. In *Proceedings of the 17th International World Wide Web Conference*, pages 477–486, Beijing, China, 2008.

[91] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 190–201, Cambridge, MA, November 2000. ACM.

[92] Paul J. Leach and Dilip C. Naik. A common internet file system (cifs/1.0) protocol. IETF Network Working Group RFC Draft, March 1997.

[93] Ronny Lempel, Yosi Mass, Shila Ofek-Koifman, Yael Petruschka, Dafna Sheinwald, and Ron Sivan. Just in time indexing for up to the second search. In *Proceedings of the 2007 International Conference on Information and Knowledge Management Systems (CIKM '07)*, pages 97–106, Lisboa, Portugal, 2007.

[94] Ronny Lempel and Shlomo Moran. Predictive caching and prefetching of query results in search engines. In *Proceedings of the 12th International World Wide Web Conference*, pages 19–28, Budapest, Hungary, 2003.

[95] Nicholas Lester, Alistair Moffat, and Justin Zobel. Fast on-line index construction by geometric partitioning. In *Proceedings of the 2005 International Conference on Information and Knowledge Management Systems (CIKM '05)*, pages 776–783, November 2005.

[96] Nicholas Lester, Alistair Moffat, and Justin Zobel. Efficient online index construction for text databases. *ACM Transactions on Database Systems*, 33(3), 2008.

[97] libkdtree++. `http://libkdtree.alioth.debian.org/`, 2008.

[98] Witold Litwin, Marie-Anna Neimat, and Donovan A. Schneider. LH*—a scalable, distributed data structure. *ACM Transactions on Database Systems*, 21(4):480–525, 1996.

[99] Fang Liu, Clement Yu, and Weiui Meng. Personalized web search by mapping user queries to categories. In *Proceedings of the 2002 International Conference on Information and Knowledge Management Systems (CIKM '02)*, pages 558–565, McLean, Virginia, 2002.

[100] Max O. Lorenz. Methods of measuring the concentration of wealth. *Publications of the American Statistical Associations*, 9:209–219, 1905.

[101] Clifford A. Lynch. Selectivity estimation and query optimization in large databases with highly skewed distributio n of column values. In *Proceedings of the 14th Conference on Very Large Databases (VLDB)*, pages 240–251, San Francisco, CA, August 1988.

[102] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 105–120, San Francisco, CA, December 2004.

[103] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[104] Udi Manber and Sun Wu. GLIMPSE: A tool to search through entire file systems. In *Proceedings of the Winter 1994 USENIX Technical Conference*, 1994.

[105] Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[106] Sergey Melink, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. Building a distributed full-text index for the web. *ACM Transactions on Information Systems*, 19(3):217–241, 2001.

[107] Michael Mesnier, Eno Thereska, Daniel Ellard, Gregory R. Ganger, and Margo Seltzer. File classification in self-* storage systems. In *Proceedings of the International Conference on Autonomic Computing (ICAC '04)*, pages 44–51, New York, NY, May 2004.

[108] metaTracker. metatracker for linux. `http://www.gnome.org/projects/tracker/`, 2008.

[109] Microsoft, Inc. Enterprise search from microsoft. `http://www.microsoft.com/Enterprisesearch/`, 2008.

[110] Microsoft, Inc. Windows search 4.0. `http://www.microsoft.com/windows/products/winfamily/desktopsearch/default.mspx`, 2009.

[111] Ethan Miller and Randy Katz. An analysis of file migration in a Unix supercomputing environment. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 421–433, January 1993.

[112] Soumyadeb Mitra, Marianne Winslett, and Windsor W. Hsu. Query-based partitioning of documents and indexes for information lifecycle management. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 623–636, June 2008.

[113] Jeffrey Clifford Mogul. *Representing Information About Files*. PhD thesis, Stanford University, March 1986.

[114] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference*, pages 43–56, Boston, MA, 2006.

[115] Kiran-Kumar Muniswamy-Reddy, Charles P. Wright, Andrew Himmer, and Erez Zadok. A versatile and user-oriented versioning file system. In *Proceedings of the Third USENIX*

Conference on File and Storage Technologies (FAST), pages 115–128, San Francisco, CA, April 2004. USENIX.

[116] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, 1988.

[117] B. Clifford Neuman. The Prospero file system: A global file system based on the virtual system model. *Computing Systems*, 5(4):407–432, 1992.

[118] B. Clifford Neumann, Jennifer G. Steiner, and Jeffrey I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 USENIX Technical Conference*, pages 191–201, Dallas, TX, 1988.

[119] M. A. Olson. The design and implementation of the Inversion file system. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 205–217, San Diego, California, USA, January 1993.

[120] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of the Freenix Track: 1999 USENIX Annual Technical Conference*, pages 183–192, Monterey, CA, June 1999.

[121] Oracle. Oracle berkeley db. `http://www.oracle.com/technology/products/berkeley-db/index.html`, 2008.

[122] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[123] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the Unix 4.2 BSD file system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles (SOSP '85)*, pages 15–24, December 1985.

[124] Yoann Padioleau and Olivier Ridoux. A logic file system. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 99–112, San Antonio, TX, June 2003.

[125] Aleatha Parker-Wood. Fast security-aware search on large scale file systems. UCSC CMPS 221 Class Project Report, March 2009.

[126] Swapnil Patil, Garth A. Gibson, Gregory R. Ganger, Julio Lopez, Milo Polte, Wattawat Tantisiroj, and Lin Xiao. In search of an api for scalable file systems: Under the table or above it? In *Proceedings of the 1st USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '09)*, 2009.

[127] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and Dave Hitz. NFS version 3: Design and implementation. In *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 137–151, 1994.

[128] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 105–120, 2005.

[129] Private NetApp, Inc. Customers. On the efficiency of modern metadata search appliances, 2008.

[130] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 89–101, Monterey, California, USA, 2002. USENIX.

[131] K. K. Ramakrishnan, Prabuddha Biswas, and Ramakrishna Karedla. Analysis of file i/o traces in commercial computing environments. In *Proceedings of the 1992 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 78–90, 1992.

[132] Berthier A. Ribeiro-Neto and Ramurti A. Barbosa. Query performance for tightly coupled distributed digital libraries. In *Proceedings of the Third ACM International Conference on Digital Libraries (DL '98)*, pages 182–190, 1998.

[133] Berthier Ribiero-Neto, Edleno S. Moura, Marden S. Neubert, and Nivio Ziviani. Efficient distributed algorithms to build inverted files. In *Proceedings of the 22nd Annual Interna-*

tional ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '99), pages 105–112, 1999.

[134] Alma Riska and Erik Riedel. Disk drive level workload characterization. In *Proceedings of the 2006 USENIX Annual Technical Conference*, pages 97–102, Boston, MA, May 2006.

[135] Stephen E. Robertson, Steve Walker, and Micheline Hancock-Beaulieu. Okapi at TREC-7. In *Proceedings of the 7th Text REtrieval Conference (TREC 1998)*, Gaithersburg, MD, 1998.

[136] John T. Robinson. The K-D-B-tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, pages 10–18, 1981.

[137] Drew Roselli, Jay Lorch, and Tom Anderson. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 41–54, San Diego, CA, June 2000. USENIX Association.

[138] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[139] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 188–201, Banff, Canada, October 2001. ACM.

[140] Antony Rowstrong and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.

[141] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 15–30. USENIX, December 2002.

[142] Brandon Salmon, Frank Hady, and Jay Melican. Learning to share: a study of sharing among home storage devices. Technical Report cmu-pdl-07-107, Carnegie Mellon University, 2007.

[143] Sandia National Laboratories. PDSI SciDAC released trace data. `http://www.cs.sandia.gov/Scalable_IO/SNL_Trace_Data/index.html`, 2009.

[144] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 110–123, December 1999.

[145] Mahadev Satyanarayanan. A study of file sizes and functional lifetimes. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP '81)*, pages 96–108, December 1981.

[146] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 231–244. USENIX, January 2002.

[147] Margo Seltzer, Greg Ganger, M. Kirk McKusick, Keith Smith, Craig Soules, and Christopher Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 18–23, June 2000.

[148] Margo Seltzer and Nicholas Murphy. Hierarchical file systems are dead. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS-XII)*, 2009.

[149] Sam Shah, Craig A. N. Soules, Gregory R. Ganger, and Brian D. Noble. Using provenance to aid in personal file search. In *Proceedings of the 2007 USENIX Annual Technical Conference*, pages 171–184, June 2007.

[150] Tracy F. Sienknecht, Richard J. Friedrich, Joseph J. Martinka, and Peter M. Friedenbach. The implications of distrubited data in a commercial environment on the design of hierarchical storage management. *Performance Evaluation*, 20(3), 1994.

[151] Craig Silverstein, Hannes Marais, Monika Henzinger, and Michael Moricz. Analysis of a very large web search engine query log. *ACM SIGIR Forum*, 33(1):6–12, 1999.

[152] H. A. Simon. On a class of skew distribution functions. *Biometrika*, 42:425–440, 1955.

[153] K. Smith and M. Seltzer. File layout and file system performance. Technical Report TR-35-94, Harvard University, 1994.

[154] Craig A. N. Soules. *Using Context to Assist in Personal File Retrieval*. PhD thesis, Carnegie Mellon University, 2006.

[155] Craig A. N. Soules and Gregory R. Ganger. Connections: Using context to enhance file search. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 119–132, New York, NY, USA, 2005. ACM Press.

[156] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)*, pages 43–58, San Francisco, CA, 2003.

[157] Craig A.N. Soules and Gregory R. Ganger. Why can't I find my files? New methods for automatic attribute assignment. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*, Sydney, Australia, 1999.

[158] Craig A.N. Soules, Kimberly Keeton, and Charles B. Morrey III. SCAN-Lite: Enterprise-wide analysis on the cheap. In *Proceedings of EuroSys 2009*, pages 117–130, Nuremberg, Germany, 2009.

[159] Spec benchmarks. http://www.spec.org/benchmarks.html.

[160] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7), 1981.

[161] Michael Stonebraker, Chuck Bear, Uğur Çetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers, and Stan Zdonik. One size fits all?—part 2: Benchmarking results. In *3rd Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 173–184, January 2007.

[162] Michael Stonebraker and Ugur Cetintemel. "One Size Fits All": An idea whose time has come and gone. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, pages 2–11, Tokyo, Japan, 2005.

[163] Michael Stonebraker and Lawrence C. Rowe. The design of POSTGRES. *ACM SIGMOD Record*, 15(2):340–355, 1986.

[164] Mike Stonebraker, Daniel 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 Conference on Very Large Databases (VLDB)*, pages 553–564, Trondheim, Norway, 2005.

[165] Mike Stonebraker, Sam Madden, Daniel 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 Conference on Very Large Databases (VLDB)*, pages 1150–1160, Vienna, Austria, 2007.

[166] Mark W. Storer, Kevin M. Greenan, Ethan L. Miller, and Kaladhar Voruganti. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–16, February 2008.

[167] Christina R. Strong. Keyword analysis of file system contents. UCSC CMPS 221 Class Project Report, March 2009.

[168] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 1–14, January 1996.

[169] Tcpdump/libpcap public repository. http://www.tcpdump.org/.

[170] Jaime Teevan, Christine Alvarado, Mark S. Ackerman, and David R. Karger. The perfect search engine is not enough: a study of orienteering behavior in directed search. In *Proceedings of the 2004 Conference on Human Factors in Computing Systems (CHI '04)*, pages 415–422. ACM Press, 2004.

[171] The Washington Post. The Saga Of the Lost Space Tapes: nasa is stumped in search for videos of 1969 moonwalk. `http://www.washingtonpost.com/wp-dyn/content/article/2007/01/30/AR2007013002065.html`, 2007.

[172] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 224–237, 1997.

[173] Anthony Tomasic, Hector Garcia-Molina, and Kurt Shoens. Incremental updates of inverted lists for text document retrieval. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 289–300, Minneaplois, MN, May 1984.

[174] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P. Wright. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage*, 4(2), May 2008.

[175] United States Congress. The health insurance portability and accountability act (hipaa). `http://www.hhs.gov/ocr/hipaa/`, 1996.

[176] United States Congress. The sarbanes-oxley act (sox). `http://www.soxlaw.com/`, 2002.

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

[178] Feng Wang, Qin Xin, Bo Hong, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Tyce T. McLarty. File system workload analysis for large scale scientific computing applications. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 139–152, College Park, MD, April 2004.

[179] Peter Webb. Digital data management needs holistic approach to obtain quantifiable results. *The American Oil & Gas Reporter*, November 2005.

[180] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–320, Seattle, WA, November 2006. USENIX.

[181] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*, Tampa, FL, November 2006. ACM.

[182] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04)*, Pittsburgh, PA, November 2004. ACM.

[183] 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)*, pages 17–33, February 2008.

[184] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 96–108, Copper Mountain, CO, 1995. ACM Press.

[185] Wireshark: Go deep. http://www.wireshark.org/.

[186] Theodore M. Wong and John Wilkes. My cache or yours? Making storage more exclusive. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 161–175, Monterey, CA, June 2002. USENIX Association.

[187] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems*, 31(1):1–38, March 2006.

[188] Zhichen Xu, Magnus Karlsson, Chunqiang Tang, and Christos Karamanolis. Towards a semantic-aware file store. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*, May 2003.

[189] Lawrence L. You, Kristal T. Pollack, and Darrell D. E. Long. Deep Store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, pages 804–815, Tokyo, Japan, April 2005.

[190] Zhihui Zhang and Kanad Ghose. hfs: A hybrid file system prototype for improving small file and metadata performance. In *Proceedings of EuroSys 2007*, pages 175–187, March 2007.

[191] Min Zhou and Alan Jay Smith. Analysis of personal computer workloads. In *Proceedings of the 7th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '99)*, pages 208–217, Washington, DC, 1999.

[192] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2):6, 2006.