Magellan: A Searchable Metadata Architecture for Large-Scale File Systems

Technical Report UCSC-SSRC-09-07
November 2009

Andrew W. Leung
aleung@cs.ucsc.edu

Ian F. Adams
iadams@cs.ucsc.edu

Ethan L. Miller
elm@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/

Magellan: A Searchable Metadata Architecture for Large-Scale File Systems

Andrew W. Leung Ian F. Adams Ethan L. Miller

Storage Systems Research Center
University of California, Santa Cruz

Abstract

As file systems continue to grow, metadata search is becoming an increasingly important way to access and manage files. However, existing solutions that build a separate metadata database outside of the file system face consistency and management challenges at large-scales. To address these issues, we developed Magellan, a new large-scale file system metadata architecture that enables the file system's metadata to be efficiently and directly searched. This allows Magellan to avoid the consistency and management challenges of a separate database, while providing performance comparable to that of other large file systems.

Magellan enables metadata search by introducing several techniques to metadata server design. First, Magellan uses a new on-disk inode layout that makes metadata retrieval efficient for searches. Second, Magellan indexes inodes in data structures that enable fast, multi-attribute search and allow *all* metadata lookups, including directory searches, to be handled as queries. Third, a query routing technique helps to keeps the search space small, even at large-scales. Fourth, a new journaling mechanism enables efficient update performance and metadata reliability. An evaluation with real-world metadata from a file system shows that, by combining these techniques, Magellan is capable of searching millions of files in under a second, while providing metadata performance comparable to, and sometimes better than, other large-scale file systems.

1. Introduction

The ever increasing amounts of data being stored in enterprise, cloud and high performance systems, is changing the way we access and manage files. As modern file systems reach petabyte-scale, locating and managing files has become increasingly difficult, leading to a trend towards searchable file systems. File system search is appealing since it is often easier to specify *what* one wants using file metadata and extended attributes rather than specifying *where* to find it [39]. Searchable metadata allows users and administrators to ask complex, ad hoc questions about the properties of the files being stored, helping them to locate, manage, and analyze their data. These needs have led to an increasing demand for metadata search in high-end computing (HEC) [20] and enterprise [13] file systems.

Unfortunately, current file systems are ill-suited for search because today's metadata designs still resemble those designed over forty years ago, when file systems contained orders of magnitude fewer files and basic namespace navigation was more than sufficient [12]. As a result, metadata searches can require brute-force namespace traversal, which is not practical at large scale. To address this problem, metadata search is implemented with a search application—a separate database of the file system's metadata—as is done in Linux (the `locate` utility), personal computers [8], and enterprise search appliances [18, 25].

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 metadata changes in the file system, a difficult challenge in a system with billions of files and constant metadata changes. Second, metadata changes must be quickly re-indexed to prevent a search from returning very inaccurate results. Keeping the metadata index and "real" file system consistent is difficult because collecting metadata changes is often slow [23, 40] and search applications are often inefficient to update [1]. 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 metadata search functionality directly into the file system. This eliminates the need to manage a secondary database, allowing metadata changes to be searched in real-time, and enabling metadata organization that corresponds to the users' need for search functionality. However, enabling metadata search within the file system has its own challenges. First, metadata must be

organized 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 [15, 31], have had difficulty addressing these challenges.

To address the problem of search in large-scale file systems, we developed a new file system metadata design, Magellan, that enables metadata search within the file system while maintaining good file system performance. Unlike previous work, Magellan does not use relational databases to enable search. Instead, its uses new query-optimized metadata layout, indexing, and update techniques to ensure searchability and high performance in a single metadata system. In Magellan, *all* metadata lookups, including directory lookups, 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.

The remainder of this paper is organized as follows. Section 2 discusses the challenges for combining metadata search and file systems and Section 3 describes related work. The Magellan design is presented in Section 4 and our prototype implementation is evaluated in Section 5. In Section 6 we discuss lessons we learned while designing Magellan and future work. We conclude in Section 7.

2. 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 metadata search has grown; this need has not been adequately met by existing approaches.

2.1 Why Do We Need Metadata Search?

Large-scale file systems make locating and managing files extremely difficult; this problem is sufficiently acute that there is increasing sentiment that "hierarchical file systems are dead" [39]. 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 relocating files for maximum efficiency that businesses must solve. Search helps to address these problems by allowing users and administrators to state what they want and locate it quickly.

Metadata search is particularly helpful because it not only helps users locate files but also provides database-like analytic queries over important attributes. Metadata attributes, which are represented as $\langle attribute, value \rangle$ pairs, include file inode fields (*e.g.*, size, owner, timestamps, *etc.*) and extended attributes (*e.g.*, document title, retention policy, and provenance). Metadata search helps users to understand the kinds of files being stored, where they are located, how they are used, and where they should belong. In previous work, we conducted a survey that found that metadata search was being used for a variety of purposes, including managing storage tiers, complying with legislation such as Sarbanes-Oxley, searching scientific data, and capacity planning [26].

2.2 Search Applications

File system search is traditionally addressed with a separate search application, such as the Linux `locate` program, Apple Spotlight [8] and Google Enterprise Search [18]. Search applications re-index file metadata 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 metadata without the need for file system modifications, making them easy to deploy onto existing systems.

These applications have been very successful on desktop and smaller scale file systems. However, they require that two separate indexes of all metadata be maintained—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) *Metadata must be replicated in the search application.* Metadata 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 metadata change 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 aware of the search application. Unfortunately, search applications are search-optimized, which often makes update performance notoriously slow [1]. 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 [18, 25], making them expensive and difficult to manage at large-scales. Modern large file systems focus on energy consumption and consolidation [6], 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 [41]; thus, accessing a file through a search application can be much less efficient than through a file system [39]. 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.

2.3 Integrating Search into the File System

We believe that a more complete solution is for the file system to organize its metadata to facilitate 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 metadata 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 [12], 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 metadata.* File systems often have metadata scattered throughout the disk [14]. Scanning the metadata for millions of files for a search can require many expensive disk seeks.

2) *Queries must quickly analyze large amounts of metadata.* Metadata must be scanned to find files that match the query. Because file systems do not directly index metadata attributes, they must use linear search or similarly slow techniques to find relevant files.

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

3. Related Work

While file systems have been hierarchically organized for over four decades, there have been many attempts to provide a more searchable interface to them. Early search applications such as `grep` and `find` made brute force search easier to use, but not faster. More recently, Diamond [23] introduced *early discard* to improve brute force search performance by quickly determining if a file was relevant to a search. Diamond used brute force search because they claimed that maintaining a separate search application presents significant challenges at large-scales.

More recently, Spyglass [26], a search application that we designed, showed that search performance can be improved with an index specially tailored for metadata search, rather than a general-purpose database. We leverage some of its index designs in Magellan. However, a major drawback was that it did not handle real-time updates which resulted in stale search results. SmartStore [22] is a similar system that indexes metadata in a distributed R-tree and uses Latent Semantic Indexing (LSI) to group correlated metadata. SmartStore also does not handle real-time updates, and the use of LSI limits its ability to perform index updates quickly.

Magellan is not the first file system to try to integrate searching directly into the file system. Semantic file systems [17, 19, 32] replaced the hierarchical namespace with a semantic, search-based one. In these file systems, queries were the main method of file access and a dynamic namespace could be built using *virtual directories*, which contained the results of a search. Semantic file systems indexed both metadata and file content and aimed to provide real-time updates. However, these systems had performance and consistency issues that Magellan can help address by providing real-time metadata search and updates. File systems such as LiFS [4] (and the Web itself) have moved away from a purely hierarchical file tree towards a more flexible interconnection scheme, but such file systems either lack high performance searchability at scale or require Google-scale data centers to answer search queries rapidly.

Other file systems replaced the file system internals with a relational database [15, 31], allowing database functionality, such as search, to be provided by the file system. However, databases were not designed to be file systems and are not a "one size fits all" solution [42]; again, these file systems experienced performance problems on both regular file accesses and metadata search.

PLDIR [30] and BeFS [16] provided better ways to internally represent metadata 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. BeFS indexed extended attributes using B+-trees. While its indexing capabilities were very basic, it did address some index update and consistency issues.

4. Magellan Design

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 Spyglass [26].

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 [10].
- 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 [44]. 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 [11] 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.

4.1 Metadata Clustering

In existing file systems, searches must read large amounts of metadata from disk since file system searches 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`.

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 [29]. 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 [14]. We illustrate this concept in the top part of Figure 1, which shows how a sub-tree can be scattered on disk.

Magellan addresses this problem by grouping inodes into large groups called *clusters*. 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

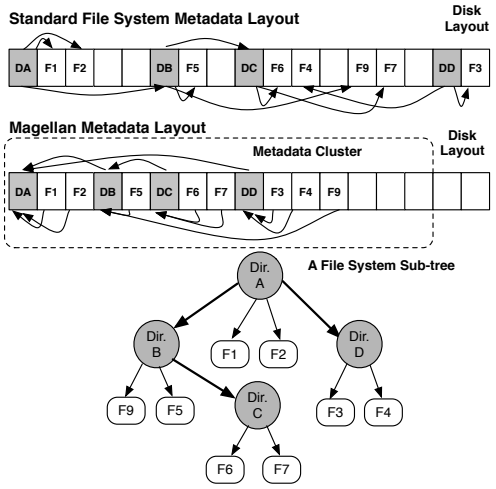


Figure 1. Metadata clustering. Each block corresponds to an inode on disk. Shaded blocks labeled 'D' are directory inodes while non-shaded blocks labeled 'F' are file inodes. In the top disk layout, the indirection between directory and file inodes causes them to be scattered across the disk. The bottom 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.

store inodes corresponding to the files and directories in the `/projects/magellan/` sub-tree. The bottom part of Figure 1 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 [14] 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 [3, 26]. 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, which exhibit similar locality in their workloads [27, 35]. Often, workloads access multiple, related directories, which clustering works well for.

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 4.2.

While clustering can improve performance by allowing fast pre-fetching 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 pre-fetching 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.

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. We plan to address these issues with a clustering algorithm that rebalances cluster distributions over time.

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 4.4); (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.

4.2 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.

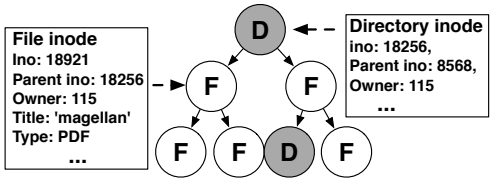


Figure 2. Inode indexing with a K-D tree. A K-D tree is shown with nodes that are directory inodes are shaded with a 'D'. File inodes are not shaded and labeled 'F'. K-D trees are organized based on attribute value not namespace hierarchy. Thus, a file inode can point to other file inodes, *etc.* The 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.

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 [9]. 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 2. Figure 2 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 2. 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. 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.

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 rebalanceing 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.

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+ [7], index inodes using just the inode number, often in a B-tree. Operations such as path resolution that perform lookups 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 lookups, avoiding K-D tree lookups 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.

4.3 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 4.1, 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* [10] to describe the contents of each cluster and to *route queries* to only the sub-trees that contain relevant metadata. 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.

4.4 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 [33, 38]. 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 [36].

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 [27], 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.

Since metadata updates are not immediately written to their primary on-disk location, the journal can grow very large. Magellan allows the journal to grow to hundreds of megabytes before requiring that clusters be flushed. Since journal writes are a significant part of update performance, staging the journal in non-volatile memory such as flash

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 1. Inode attributes used. The attributes that inodes contained in our experiments.

memory or phase change memory could significantly boost performance.

5. Experimental Results

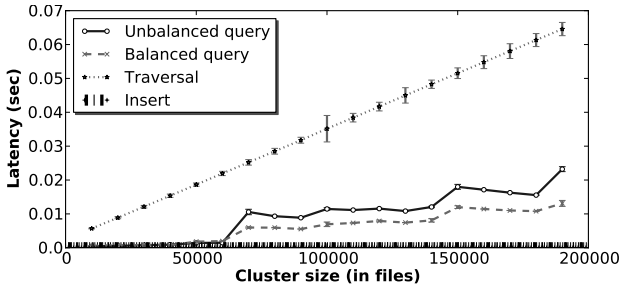
Our evaluation seeks to examine the following questions: (1) How does Magellan's metadata indexing impact performance? (2) How does our journaling technique affect metadata updates? (3) Does metadata clustering improve disk utilization? (4) How does our prototype's metadata performance compare to other 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.

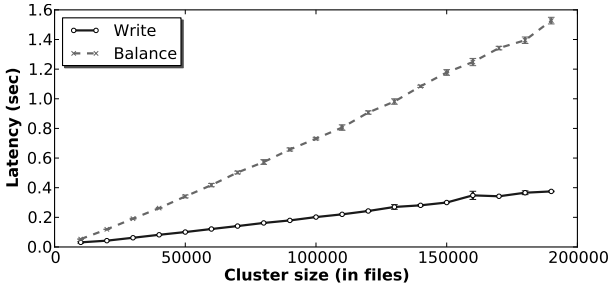
5.1 Implementation Details

We implemented our prototype as the metadata server (MDS) for the Ceph parallel file system [44], for several reasons. First, parallel file systems often handle metadata and data separately [11, 44]: 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 [45], freeing Magellan from the need to manage data block pointers. 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. The K-D tree in each cluster is implemented using `libkdtree++` [28], version 0.7.0. Each inode has eleven attributes that are indexed, listed in Table 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 metadata cache is 100 KB in size. 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



(a) Query and insert performance.



(b) Write and optimize performance.

Figure 3. Cluster indexing performance. Figure 3(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 3(b) shows latencies for K-D tree rebalancing and disk writes. Rebalancing is slower because it is requires $O(N \times \log N)$ time.

present a significant implementation barrier, and none significantly impact performance; we will implement them in the future.

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.

5.2 Microbenchmarks

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

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 3(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 [2]. We used 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 $O(kN^{1-1/k})$ time, where N is the number of files, and k is the dimensionality, meaning that performance increasingly degrades with cluster size.

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 us. 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 3(b) shows latencies for writing a cluster to disk and rebalancing, the two major steps performed when a dirty cluster is written to disk. Surprisingly, rebalancing is the more significant of the two steps, taking $3-4\times$ longer than writing to disk. The K-D tree rebalancing algorithm takes $O(N \times \log N)$ time, accounting 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.

Update Performance. To evaluate how our cluster-based journaling impacts update performance, we used a create benchmark that creates between 100,000 and 2,000,000 files, and measured the throughput at various sizes. To do this, we used metadata traces that we collected from three storage servers deployed at NetApp [27]. We used different traces because each has different namespace organizations that impact performance (*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 (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.

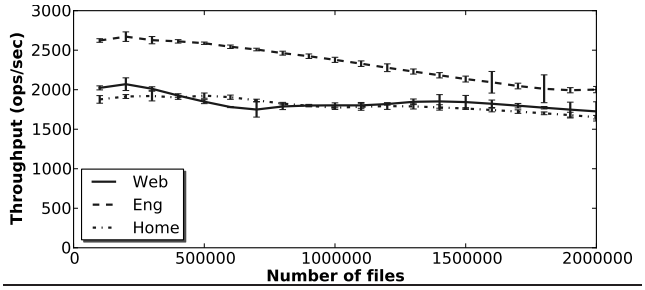
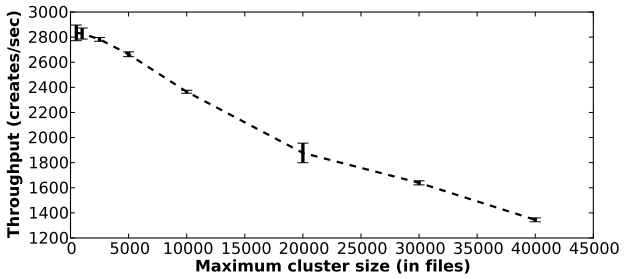


Figure 4. Create performance. The throughput (creates/second) is shown for various system sizes. Magellan's update mechanism keeps create throughput high because disk writes are mostly to the end of the journal, which yields good disk utilization. Throughput drops slightly at larger sizes because more time is spent searching clusters.

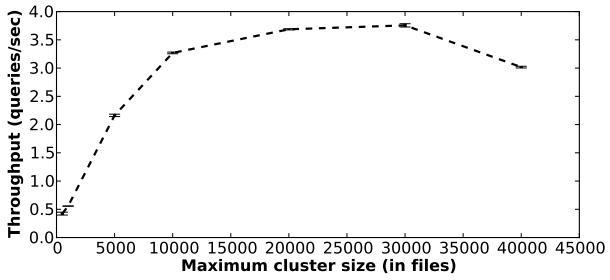
Figure 4 shows the throughput averaged over five runs and standard deviations as the number of creates increases. We find that, in general, throughput is very high, between 1,500 and 2,500 creates per second, because of Magellan's cluster-based journaling. Each create appends an update entry to the on-disk 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; again, though, the higher rate of false positives causes more K-D tree searches.

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.



(a) Create latencies



(b) Query latencies.

Figure 5. Metadata clustering. Figure 5(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(b) shows that query performance is worse for small and large sizes.

Figure 5(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 metadata 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(b) shows that query performance scales quite differently from create performance. We used a simple query that represented a user search for a file she owns with a particular name (*e.g.,* `filename` equal to `mypaper.pdf` and `owner id` equal to `3704`). We 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 needed to read the metadata needed. 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.

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 2. Metadata workload details.

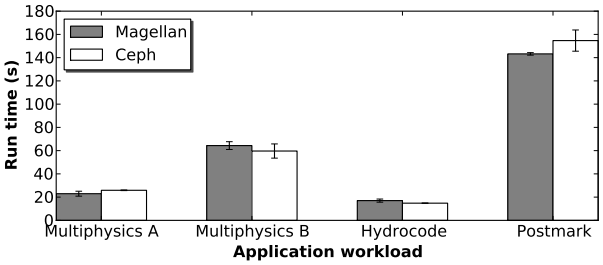


Figure 6. Metadata workload performance comparison. Magellan is compared to the Ceph metadata server using four different metadata workloads. In all cases, both provide comparable performance. Performance differences are often do to K-D tree utilization.

5.3 Macrobenchmarks

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

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 [37] and the other is the Postmark [24] benchmark. Table 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. While the benchmarks are not large enough to evaluate all aspects of file system performance (many common metadata benchmarks are not [43]), 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 6 shows the run times and standard errors averaged over five runs, with the corresponding throughputs shown in Table 3. Total run times are comparable for both, showing that Magellan is able to achieve similar file system

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

Table 3. Metadata throughput (ops/sec). The Magellan and Ceph throughput for the four workloads.

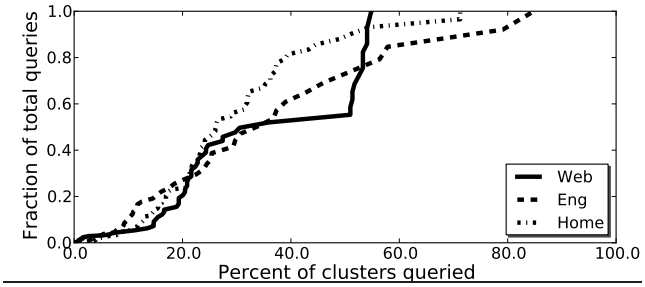


Figure 8. Fraction of clusters queried. A CDF of the fraction of clusters queried for query set 1 on our three data sets is shown. Magellan is able to leverage Bloom filters to exploit namespace locality and eliminate many clusters from the search space.

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 files. Table 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 out 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.

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 inodes, not rows in a database table). This validates an important goal of our design: address issues with search performance while maintaining many aspects that current metadata designs do well.

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 use cases. The queries and the attributes used are given in Table 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.

Figure 8 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 size increases. In fact, all queries across all entire traces are less six seconds, with the exception of several in the Home trace that were between eight and fifteen seconds. A key reason is that Magellan is able to leverage namespace locality by using Bloom filters to eliminate a large fraction of the clusters from the search space. Figure 8 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.

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 7(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.

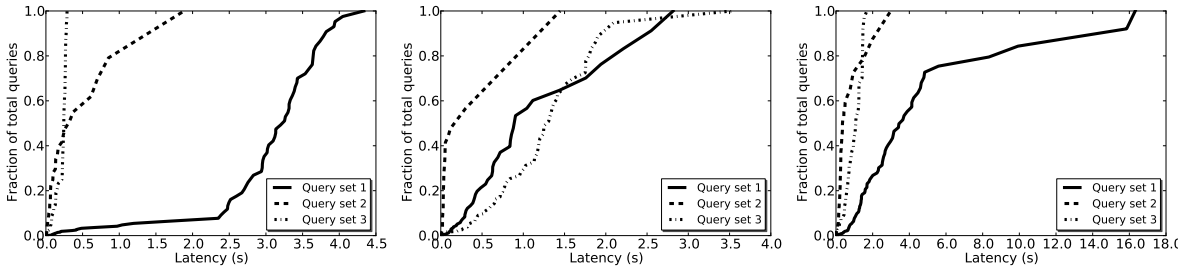
6. Discussion and Future Work

Our experience in designing, building and testing Magellan has provided valuable insights regarding current and future searchable file system designs. Early on in our design, we noticed how close file systems already are to being able to provide metadata search. Many of Magellan's designs, such as clustering metadata nearby in the namespace on disk, are already to done to some degree by file systems. We believe that this shows there is a low barrier to enabling search in more file systems.

While K-D trees introduce some design challenges such as rebalancing, and large cluster performance, we were actually surprised by how effective they proved to be. Given that they were neither designed for file systems nor previously used in file systems, they turned out to make inode indexing relatively straightforward. We are now looking at new benefits other structure can provide. For example, FastBit [46]

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 4. Query Sets. A summary of the searches used to generate our evaluation query sets.



(a) Web server. (b) Engineering server. (c) Home server.

Figure 7. 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.

can provide high compression ratios and K-D-B-trees [34] can provide partially in-memory K-D trees.

A major challenge that we do not address, but are examining, is the metadata search interface. Magellan's search interface, which used complex, SQL-like queries was cumbersome and it is unlikely users will want to frequently access their data this way. However, a simple Google-like search box may be too simplistic. If search is to become effective enough for daily use a good interface is needed, perhaps similar to that provided by QUASAR [5] or PQL [21].

Content-based search supporting queries such as "find all files containing the term `eurosys`" shares some similarities with metadata search: searching file content requires reading file data, extracting and analyzing keywords, and indexing them, which file systems do not do and which can be difficult at large-scales. However, combining metadata search with the file system is more straightforward because file systems already store and index metadata. Regardless, content-based search provides a powerful tool, and we are looking at new ways to enable improved content search in large-scale file systems.

7. Conclusions

The rapid growth in data volume is changing how we access and manage our files. Large-scale systems increasingly require metadata 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 metadata search solutions. We believe that a better approach is to build metadata search directly into the file system itself.

In this paper, 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. 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's search-optimized design does have limitations, it demonstrates that metadata 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.

References

[1] ABADI, D. J., MADDEN, S. R., AND HACHEM, N. 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] AGRAWAL, N., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Generating realistic *impressions* for file-system benchmarking. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)* (Feb. 2009), pp. 125–138.

[3] AGRAWAL, N., BOLOSKY, W. J., DOUCEUR, J. R., AND LORCH, J. R. A five-year study of file-system metadata. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)* (Feb. 2007), pp. 31–45.

[4] AMES, S., BOBB, N., GREENAN, K. M., HOFMANN, O. S., STORER, M. W., MALTZAHN, C., MILLER, E. L., AND BRANDT, S. A. 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.

[5] AMES, S., MALTZAHN, C., AND MILLER, E. L. Quasar: A scalable naming language for very large file collections. Tech. Rep. UCSC-

SSRC-08-04, University of California, Santa Cruz, Oct. 2008.

[6] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., AND VASUDEVAN, V. Fawn: A fast array of wimpy nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)* (2009).

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

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

[9] BENTLEY, J. L. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (Sept. 1975), 509–517.

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

[11] CARNS, P. H., LIGON, W. B., ROSS, R. B., AND THAKUR, R. PVFS: a parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference* (Atlanta, GA, Oct. 2000), pp. 317–327.

[12] DALEY, R., AND NEUMANN, P. A general-purpose file system for secondary storage. In *Proceedings of the Fall Joint Computer Conference, Part I* (1965), pp. 213–229.

[13] ENTERPRISE STRATEGY GROUPS. ESG Research Report: storage resource management on the launch pad, 2007.

[14] GANGER, G. R., AND KAASHOEK, M. F. Embedded inodes and explicit groupings: Exploiting disk bandwidth for small files. In *Proceedings of the 1997 USENIX Annual Technical Conference* (Jan. 1997), USENIX Association, pp. 1–17.

[15] GEHANI, N., JAGADISH, H., AND ROOME, W. D. Odefs: A file system interface to an object-oriented database. In *Proceedings of the 20th Conference on Very Large Databases (VLDB)* (Santiago de Chile, Chile, 1984), pp. 249–260.

[16] GIAMPALO, D. *Practical File System Design with the Be File System*, 1st ed. Morgan Kaufmann, 1999.

[17] GIFFORD, D. K., JOUVELOT, P., SHELDON, M. A., AND O'TOOLE, JR., J. W. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)* (Oct. 1991), ACM, pp. 16–25.

[18] GOOGLE, INC. Google enterprise. `http://www.google.com/enterprise/`, 2008.

[19] GOPAL, B., AND MANBER, U. Integrating content-based access mechanisms with hierarchical file systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)* (Feb. 1999), pp. 265–278.

[20] GRIDER, G., NUNEZ, J., BENT, J., ROSS, R., WARD, L., POOLE, S., FELIX, E., SALMON, E., AND BANCROFT, M. Coordinating government funding of file system and I/O research through the High End Computing University Research Activity. *Operating Systems Review 43*, 1 (Jan. 2009).

[21] HOLLAND, D. A., BRAUN, U., MACLEAN, D., MUNISWAMY-REDDY, K.-K., AND SELTZER, M. I. A data model and query language suitable for provenance. In *Proceedings of the 2008 International Provenance and Annotation Workshop (IPAW)* (June 2008).

[22] HUA, Y., JIANG, H., ZHU, Y., FENG, D., AND TIAN, L. SmartStore: A new metadata organization paradigm with metadata semantic-awareness for next-generation file systems. In *Proceedings of SC09* (Portland, OR, Nov. 2009).

[23] HUSTON, L., SUKTHANKAR, R., WICKREMESINGHE, R., SATYANARAYANAN, M., GANGER, G. R., RIEDEL, E., AND AILAMAKI, A. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST)* (San Francisco, CA, Apr. 2004), USENIX, pp. 73–86.

[24] KATCHER, J. PostMark: a new file system benchmark. Technical Report TR-3022, NetApp, Sunnyvale, CA, 1997.

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

[26] LEUNG, A., SHAO, M., BISSON, T., PASUPATHY, S., AND MILLER, E. L. Spyglass: Fast, scalable metadata search for large-scale storage systems. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)* (Feb. 2009), pp. 153–166.

[27] LEUNG, A. W., PASUPATHY, S., GOODSON, G., AND MILLER, E. L. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the 2008 USENIX Annual Technical Conference* (June 2008).

[28] libkdtree++. `http://libkdtree.alioth.debian.org/`, 2009.

[29] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A fast file system for UNIX. *ACM Transactions on Computer Systems 2*, 3 (Aug. 1984), 181–197.

[30] MOGUL, J. C. *Representing Information About Files*. PhD thesis, Stanford University, Mar. 1986.

[31] OLSON, M. A. The design and implementation of the Inversion file system. In *Proceedings of the Winter 1993 USENIX Technical Conference* (San Diego, California, USA, Jan. 1993), pp. 205–217.

[32] PADIOLEAU, Y., AND RIDOUX, O. A logic file system. In *Proceedings of the 2003 USENIX Annual Technical Conference* (San Antonio, TX, June 2003), pp. 99–112.

[33] PRABHAKARAN, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Analysis and evolution of journaling file systems. In *Proceedings of the 2005 USENIX Annual Technical Conference* (2005).

[34] ROBINSON, J. T. 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* (1981), pp. 10–18.

[35] ROSELLI, D., LORCH, J., AND ANDERSON, T. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference* (San Diego, CA, June 2000), USENIX Association, pp. 41–54.

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

[37] SANDIA NATIONAL LABORATORIES. PDSI SciDAC released trace data. `http://www.cs.sandia.gov/Scalable_IO/SNL_Trace_Data/index.html`, 2009.

[38] SELTZER, M., GANGER, G., MCKUSICK, M. K., SMITH, K., SOULES, C., AND STEIN, C. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Proceedings of the 2000 USENIX Annual Technical Conference* (June 2000), pp. 18–23.

[39] SELTZER, M., AND MURPHY, N. Hierarchical file systems are dead. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS-XII)* (2009).

[40] SOULES, C. A., KEETON, K., AND III, C. B. M. SCAN-Lite: Enterprise-wide analysis on the cheap. In *Proceedings of EuroSys 2009* (Nuremberg, Germany, 2009).

[41] STONEBRAKER, M. Operating system support for database management. *Communications of the ACM 24*, 7 (1981).

[42] STONEBRAKER, M., AND CETINTEMEL, U. "One Size Fits All": An idea whose time has come and gone. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)* (Tokyo, Japan, 2005), pp. 2–11.

[43] TRAEGER, A., ZADOK, E., JOUKOV, N., AND WRIGHT, C. P. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage 4*, 2 (May 2008).

[44] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)* (Seattle, WA, Nov. 2006), USENIX.

[45] WEIL, S. A., BRANDT, S. A., MILLER, E. L., AND MALTZAHN, C. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)* (Tampa, FL, Nov. 2006), ACM.

[46] WU, K., OTOO, E. J., AND SHOSHANI, A. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems 31*, 1 (Mar. 2006), 1–38.