Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems

Andrew W. Leung[*]   Minglong Shao[†]   Timothy Bisson[†]   Shankar Pasupathy[†]   Ethan L. Miller[*]

[*]*University of California, Santa Cruz*   [†]*NetApp*

{*aleung, elm*}*@cs.ucsc.edu*   {*minglong, tbisson, shankarp*}*@netapp.com*

Abstract

The scale of today's storage systems has made it increasingly difficult to find and manage files. To address this, we have developed Spyglass, a file metadata search system that is specially designed for large-scale storage systems. Using an optimized design, guided by an analysis of real-world metadata traces and a user study, Spyglass allows fast, complex searches over file metadata to help users and administrators better understand and manage their files.

Spyglass achieves fast, scalable performance through the use of several novel metadata search techniques that exploit metadata search properties. Flexible index control is provided by an index partitioning mechanism that leverages namespace locality. Signature files are used to significantly reduce a query's search space, improving performance and scalability. Snapshot-based metadata collection allows incremental crawling of only modified files. A novel index versioning mechanism provides both fast index updates and "back-in-time" search of metadata. An evaluation of our Spyglass prototype using our real-world, large-scale metadata traces shows search performance that is 1-4 orders of magnitude faster than existing solutions. The Spyglass index can quickly be updated and typically requires less than 0.1% of disk space. Additionally, metadata collection is up to $10\times$ faster than existing approaches.

1   Introduction

The rapidly growing amounts of data in today's storage systems makes finding and managing files extremely difficult. Storage users and administrators need to efficiently answer questions about the properties of the files being stored in order to properly manage this increasingly large sea of data. Metadata search, which involves indexing file metadata such as inode fields and extended attributes, can help answer many of these questions [26].

Metadata search allows point, range, top-k, and aggregation search over file properties, facilitating complex, ad hoc queries about the files being stored. 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. Efficiently answering these questions can greatly improve how user and administrator manage files in large-scale storage systems.

Unfortunately, fast and efficient metadata search in large-scale storage systems is difficult to achieve. Both customer discussions [37] and personal experience have shown that existing enterprise search tools that provide metadata search [4, 14, 17, 21, 30] are often too expensive, slow, and cumbersome to be effective in large-scale systems. *Effective* metadata search must meet several requirements. First, it must be able to quickly gather metadata from the storage system. We have observed commercial systems that took 22 hours to crawl 500 GB and 10 days to crawl 10 TB. Second, search and update must be fast and scalable. Existing systems typically index metadata in a general-purpose DBMS. However, DBMSs are not a perfect fit for metadata search, which can limit their performance and scalability in large-scale systems. Third, resource requirements must be low. Existing tools require dedicated CPU, memory, and disk hardware, making them expensive and difficult to integrate into the storage system. Fourth, the search interface must be flexible and easy to use. Metadata search enables complex file searches that are difficult to ask with existing file system interfaces and query languages. Fifth, search results must be secure; many existing systems either ignore file ACLs or significantly degrade performance to enforce them.

To address these issues, we developed Spyglass, a novel metadata search system that exploits file metadata properties to enable fast, scalable search that can be em-

bedded within the storage system. To guide our design, we collected and analyzed file metadata snapshots from real-world storage systems at NetApp and conducted a survey of over 30 users and IT administrators. Our design introduces several new metadata search 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 used to compactly describe 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. Spyglass does not currently address search interface or security, which are left to future work.

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 Spyglass can leverage file metadata properties to improve how files are managed in large-scale storage systems.

This remainder of this paper is organized as follows. Section 2 provides additional metadata search motivation and background. Section 3 presents the Spyglass design. Our prototype is evaluated in Section 4. Related work is discussed in Section 5, with future work and conclusions in Section 6.

2 Background

This section describes and motivates the use of file metadata search and includes a discussion of real-world query and metadata characteristics.

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.

The ability to search file metadata facilitates complex queries on the properties of files in the storage system, helping 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, finding which files to migrate to tape may involve searching file size, access time, and owner metadata attributes, allowing administrators to decide on and enforce their management policies. Metadata search also helps users locate misplaced files, manage their storage space, and track file changes. 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 [12].

To better understand metadata search needs, we surveyed over 30 large scale storage system users and administrators. 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 1 provides examples of some popular use cases and the metadata attributes searched.

2.2 Efficient Metadata Search

Providing efficient metadata search in large-scale storage systems is a challenge. While a number of commercial file metadata search systems exist today [4, 14, 17, 21, 30], these systems focus on smaller scales (*e.g.,* up to tens of millions of files) and are often too slow, resource intensive, and expensive to be effective for large-scale systems. To be effective at large scales, file metadata search must provide the following:

1) *Minimal resource requirements.* Metadata search should not require additional hardware. It should be embedded within the storage system and close to the files it indexes while not degrading system performance. Most existing systems require dedicated CPU, memory, and disk hardware, making them expensive and hard to deploy, and limiting their scalability.

2) *Fast metadata collection.* Metadata changes must be periodically collected from millions to billions of files without exhausting or slowing the storage system. Existing crawling methods are slow and can tax system resources. Hooks to notify systems of file changes can add overhead to important data paths.

3) *Fast and scalable index search and update.* Searches must be fast, even as the system grows, or usability may

File Management Question	Metadata Search Query
Which files can I migrate to tape?	**size** $>$ `50 GB`, **atime** $>$ `6 months`.
How many duplicates of this file are in my home directory?	**owner** $=$ `john`, **datahash** $=$ `0xE431`, **path** $=$ `/home/john`.
Where are my recently modified presentations?	**owner** $=$ `john`, **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** $=$ `john`.
How much storage do these users and applications consume?	Sum **size** where **owner** $=$ `john`, **type** $=$ `database`

Table 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 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.*

suffer. Updates must allow fast periodic re-indexing of metadata. However, existing systems typically rely on general-purpose relational databases (DBMSs) to index metadata. For example, Microsoft's enterprise search indexes metadata in their Extensible Storage Engine (ESE) database [30]. Unfortunately, DBMSs often use heavyweight locking and transactions that add overhead even when disabled [43]. Additionally, their designs make significant trade-offs between search and update performance [1]. DBMSs also assume abundant CPU, memory, and disk resources. Although standard DBMSs have benefited from decades of performance research and optimizations, such as vertical partitioning [23] and materialized views, their designs are not a perfect fit for metadata search. This is not a new concept; the DBMS community has argued that general-purpose DBMSs are not a "one size fits all solution" [9, 42, 43], instead saying that application-specific designs are often best.

4) *Easy to use search interface.* Most systems export simple search APIs. However, recent research [3] has shown that specially designed interfaces that can provide an expressive and easy to use query capabilities can greatly improve search experience.

5) *Secure search results.* Search results must not allow users to find or access restricted files [10]. Existing systems either ignore security or enforce it at a significant cost to performance.

We designed Spyglass to address these challenges in large-scale storage systems. 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; interface and security designs are left to future work.

2.3 Metadata Search Properties

To understand metadata search properties, we analyzed results from our user survey and real-world metadata snapshot traces collected from storage servers at NetApp. We then used this analysis to guide our Spyglass design.

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 2: *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.*
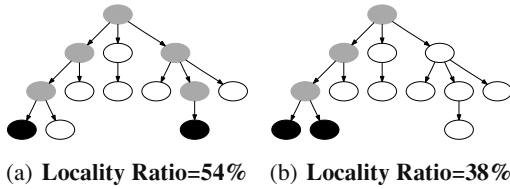
Attribute	Description	Attribute	Description
inumber	inode number	owner	file owner
path	full path name	size	file size
ext	file extension	ctime	change time
type	file or directory	atime	access time
mtime	modification time	hlink	hard link #

Table 3: *Attributes used. We analyzed the fields in the inode structure and extracted **ext** values from **path**.*

Search Characteristics. 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. 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.

Metadata Characteristics. We collected metadata snapshot traces from three storage servers at NetApp. Our traces—Web, Eng, and Home—are described in Table 2. Table 3 describes the metadata attributes that we analyzed. NetApp servers support extended attributes, though they were rarely used in these traces. We found two key properties in these traces: metadata has *spatial locality* and highly *skewed distributions* of values.

Spatial locality means that attribute values are clustered in the namespace (*i.e.,* occurring in relatively few directories). For example, `john`'s files reside mostly in

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

Figure 1: *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.*

the /home/john sub-tree, not scattered evenly across the namespace. Spatial locality comes from the way that users and applications organize files in the namespace, and has been noted in other file system studies [2, 25]. 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 1. 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. Table 4 shows the locality ratios for the 32 most frequently occurring values for various attributes (ext, size, owner, ctime, mtime) in each trace. Locality ratios are less than 1% for all attributes, meaning that 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.

Utilizing spatial locality can help prune a query's search space by identifying only the parts of the namespace that contain a metadata value, eliminating a large number of files to search. Unfortunately, most general-purpose DBMSs treat path names as flat string attributes, making it difficult for them to utilize this information, instead typically requiring them to consider *all* files for a search no matter its locality.

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 [2, 11]. Figures 2(a) and 2(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 the power law distribution [40]. 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 2(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 2(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 john and many files with ext pdf, but there are often over an order of magnitude fewer files with *both* owner john and ext pdf.

These distribution properties show that multi-attribute searches will significantly reduce the number of query results. 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 [28]; for example, they may retrieve all of john's files to find the few that are john'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.
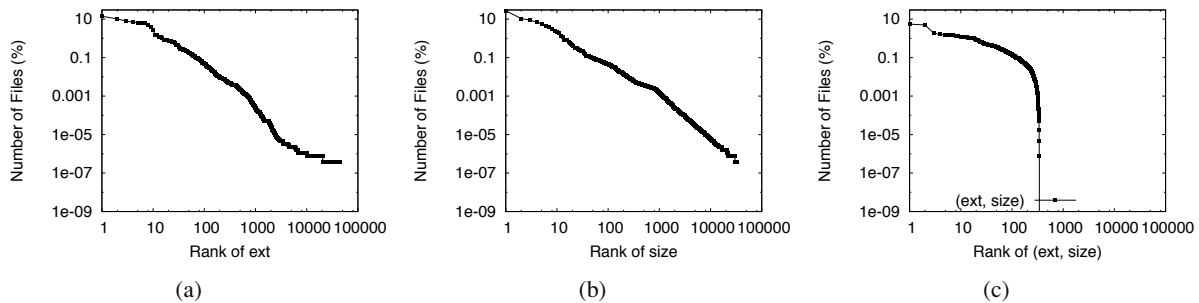
3 Spyglass Design

Spyglass uses several novel techniques that exploit the metadata search properties discussed in Section 2 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* [13] 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 3: the Spyglass index, which stores metadata and serves queries, and a crawler that extracts metadata from the storage system.
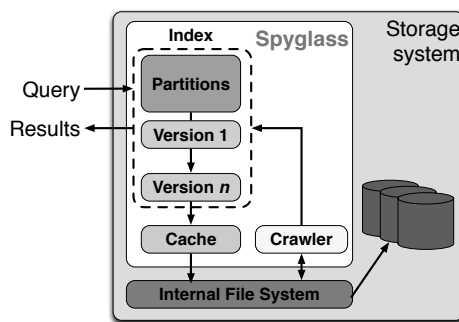
3.1 Hierarchical Partitioning

To exploit metadata locality and improve scalability, the Spyglass index is partitioned into a collection of separate, smaller indexes, which we call hierarchical partitioning. Hierarchical partitioning is based on the storage system's namespace and encapsulates separate parts of the namespace into separate partitions, thus allowing more flexible, finer grained control of the index. Similar partitioning strategies are often used by file systems to distribute the namespace across multiple machines [35, 44].

	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 4: *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.*

(a)   (b)   (c)

Figure 2: *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 2(a) and 2(b) indicate a Zipf-like distribution, while the flatter curve in Figure 2(c) indicates a more even distribution.*

Figure 3: *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.*
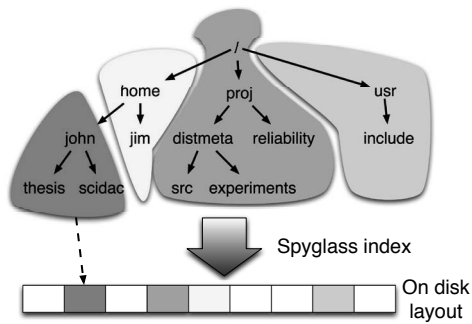
Each of the Spyglass partitions is stored sequentially on disk, as shown in Figure 4. Thus, unlike a DBMS, which stores records adjacently on disk using their row or column order, Spyglass groups records nearby in the namespace together on disk. This approach improves performance since the files that satisfy a query are often clustered in only a portion of the namespace, as shown by our observations in Section 2. For example, a search of the storage system for john'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 hierarchical 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. Since partitions are stored sequentially on disk, searches can usually be satisfied with only a few small sequential disk reads. Also, sub-trees often grow at a slower rate than the system as a whole [2, 25], 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 en-

Figure 4: *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.*

tire 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 [15], 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 [5] and file system access patterns [25] 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 frequently accessed partitions in-memory, ensuring high performance for common queries and efficient cache utilization.

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 [7], 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 [45] provides high index compression, Berkeley DB [34] provides transactional storage, cache-oblivious B-trees [6] improve update time, and K-D-B-trees [38] 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 3.2 describes techniques to avoid continuous updates.
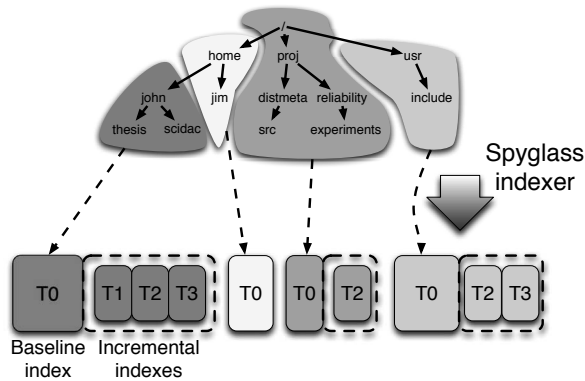
Partition Metadata. Partition metadata contains information about the files in the partition, including paths of indexed sub-trees, file statistics, signature files, and version information. File statistics, such as file counts and minimum and maximum values, are kept to answer

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 3.2, manages partition versions. Signature files are used to determine if the partition contains files relevant to a query.

Signature files [13] 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 [8]. 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 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. 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. Due to spatial locality, most searches can eliminate many partitions, reducing the number of disk accesses and processing a query must perform.

Due to 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. 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. 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.

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

Figure 5: *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.*

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.

3.2 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 index structures 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 [39, 41].

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 5. 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 [39], which maintain full copies for each version. Changes consist of 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 5. 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 [39]. Since file metadata in different parts of the file system change at different rates [2, 25], 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 john 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 john), 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 john). 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 [41], which rolls version changes backwards rather than forwards.
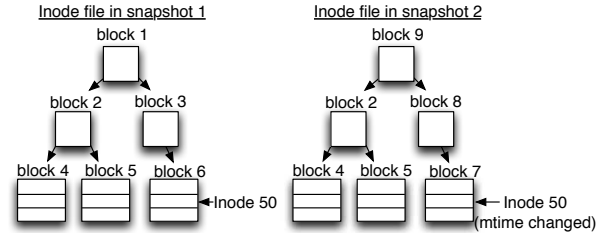
Managing Versions. Over time, older versions tends to decrease in value and should be removed to reduce search overhead and save space. Spyglass provides two efficient techniques for managing partition versions: *version collapsing* and *version checkpointing*. Version collapsing applies each partition's 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. Version checkpointing allows an index to be saved to disk as a new copy to preserve an important landmark version of the index.

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-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.

3.3 Collecting Metadata Changes

The Spyglass crawler takes advantage of NetApp Snapshot$^{\text{TM}}$ technology in the NetApp WAFL $^{\text{®}}$ file system [19] 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.

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 6. 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

Figure 6: *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.*

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` [22]), requires hooks into critical code paths, potentially impacting performance. A changelog, 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.

3.4 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 design is left to future work.

4 Experimental Evaluation

We evaluated our Spyglass prototype to determine how well our design addresses the metadata search challenges described in Section 2 for varying storage system sizes. To do this, 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.

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.
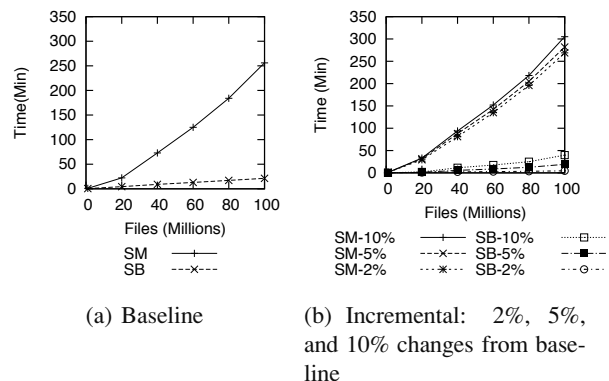
During the update process, partitions are buffered in-memory and written sequentially to disk when full; each is stored in a separate file. K-D trees were implemented using `libkdtree++` [27]. Signature file bit-arrays are about 2 KB, but *hierarchical* signature files are only 100 bytes, ensuring that 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.

Experimental Setup. We evaluated performance using our real-world metadata traces described in Table 2. 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 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.
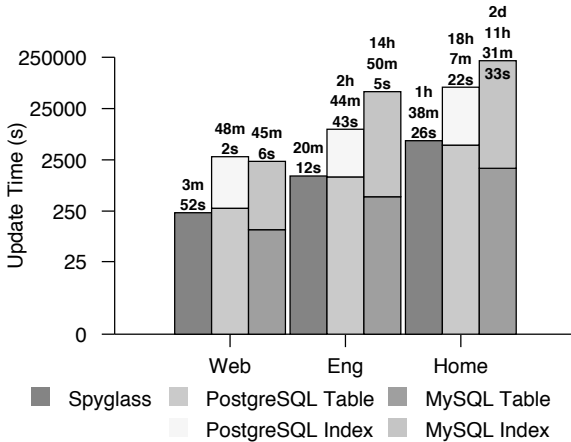
We also evaluated the performance of two popular relational DBMSs, PostgreSQL and MySQL, which serve as relative comparison points to DBMS-based solutions used in other metadata search systems. The goal of our comparison is to provide some context to frame our Spyglass evaluation, not to compare performance 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 Section 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. 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.

(a) Baseline

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

Figure 7: *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.*

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 ex-
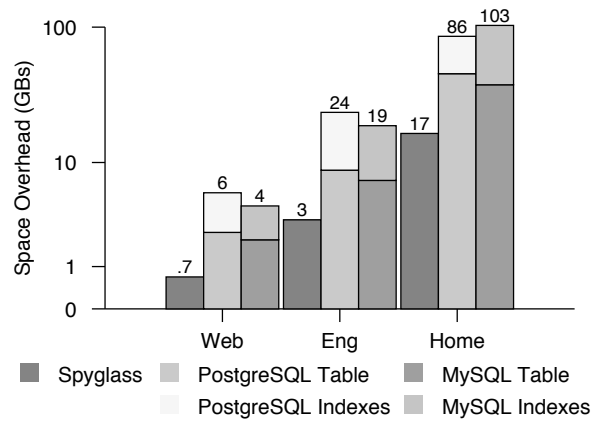
Figure 8: *Update performance. The time required to build an initial baseline index shown on a log-scale. Spyglass updates quickly and scales linearly because updates are written to disk mostly sequentially.*

Figure 9: *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.*

tract metadata. Figure 7(a) shows the performance of a baseline crawl of all file metadata. Our snapshot based crawler is up to 10× 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 7(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 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.

Update Performance. Figure 8 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× and 44× 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 data copies. As a result, DBMS updates with our Home trace can take a day or
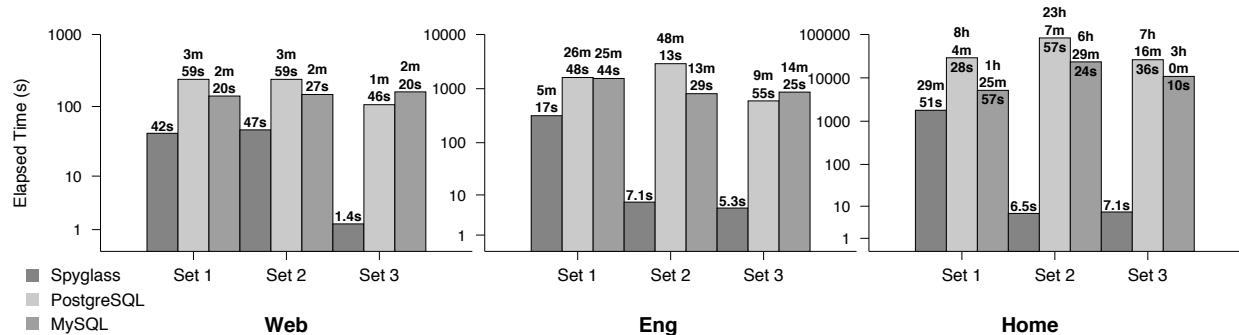
more; however, approaches such as cache-oblivious B-trees [6] may be able to reduce this gap.

Space Overhead. Figure 9 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×–8× lower than in our references DBMSs because they require space to store the base table and index structures. Figure 9 shows that building index structures can more the double the total space requirements.

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 5. 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 `john`'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.

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

Figure 10: *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.*

Figure 10 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, 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.
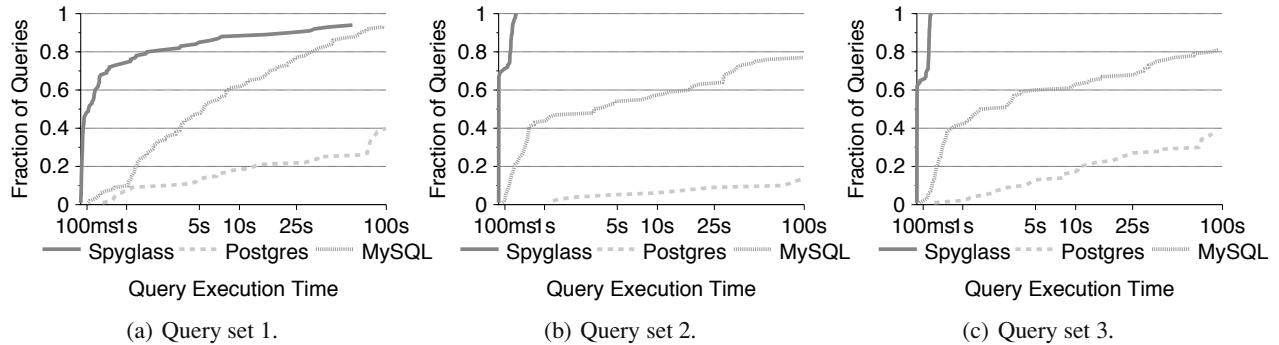
Using the results from Figure 10, we calculated query throughput, shown in Table 6. Query throughput (queries per second) provides a normalized view of our results and the query loads that can be achieved. Spyglass achieves throughput of multiple queries per second in all but two cases; in contrast, 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 11 shows an alternate view of performance; a cumulative distribution function (CDF) of query execution 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 10. These queries take longer because they must read many partitions from disk, either because few were previously cached or many partitions are searched.

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 5, 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.
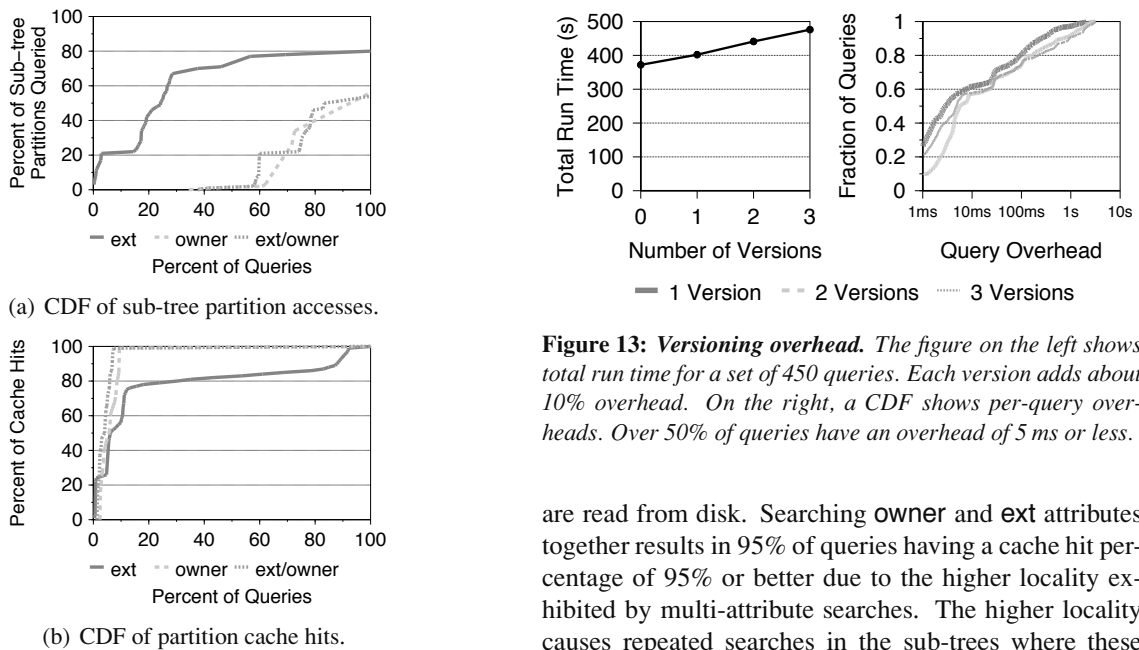
Figure 12(a) shows a CDF of the fraction of partitions searched. Searching more partitions often increases the amount of data that must be read from disk, which decreases performance. We see that 50% of searches using just the **ext** attribute reference fewer than 75% of par-

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 6: *Query throughput. We use the results from Figure 10 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.*

(a) Query set 1. (b) Query set 2. (c) Query set 3.

Figure 11: *Query execution times. A CDF of query set execution times for the Eng trace. In Figures 11(b) and 11(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 12: *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.*

titions. 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 12(b) shows a CDF of cache hit percentages for the same set of queries. Higher cache hit percentages means that fewer partitions

Figure 13: *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.*

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.

Versioning Overhead. To measure the search overhead added by partition versioning, we generated 450 queries based on query 1 from Table 5 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 13 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 13 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.

5   Related Work

Spyglass seeks to improve how file systems manage growing volumes of data, which has been an important challenge and an active area of research for over two decades. A significant amount of work has looked at how file systems can improve file naming and organization by leveraging file attributes. The Semantic File System [16] utilized file $\langle attribute, value \rangle$ pairs to dynamically construct a namespace based on queries rather than use a standard hierarchical namespace. Virtual directories allowed queries to be integrated directly into the namespace as a directory containing search results. The Hierarchy and Content (HAC) [18] file system looked as how Semantic File System concepts could be applied to a hierarchical namespace, providing users with a new naming mechanism without requiring them to forgo traditional hierarchies. These and similar systems [32, 36] focus on how users name and view files, though they do not focus on how files are actually indexed and searched, thereby potentially limiting their performance and scalability. While Spyglass does not provide higher level naming semantics, it is the first to address the challenge of scalable file metadata indexing and search, allowing it to potentially be used as the underlying indexing method for such file systems.

Spyglass focuses on how to exploit file metadata properties to improve search performance and scalability, though it is not the first to look at how new indexing structures improve file retrieval. Inversion [33] used a general-purpose DBMS as the core file system structure, rather than traditional file system inode and data layouts. Inversion used several PostgreSQL tables to store both file metadata and data, allowing the file system to benefit from database transaction and recovery support and allowing metadata and data to be queried. Like Spyglass, Inversion provides ad hoc metadata query functionality, though it focuses on allowing file systems to leverage database functionality rather than on query performance.

However, a number of new index designs have been proposed to improve various aspects of file system search. GLIMPSE [29] 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. Like Spyglass, 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. Similarly, Diamond [20] eliminated disk space requirements by using a mechanism to improve the speed of brute force searches instead of maintaining an index. A technique called Early Discard allowed files that are irrelevant to the search to be rejected as early as possible, helping to reduce the search space. Early Discard used application-specific "searchlets" to determine when a file is irrelevant to a given query. Geometric partitioning [24] 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 [31] 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.

6   Conclusions and Future Work

As storage systems have become larger, finding and managing files has become increasingly difficult. To address this problem we presented Spyglass, a metadata search system that improves file management by allowing complex, ad hoc queries over file metadata. Spyglass introduces several novel indexing techniques that improve metadata crawling, search, and update performance by exploiting metadata properties. Our evaluation shows that Spyglass has up to 1–4 orders of magnitude faster search performance then existing designs.

We plan on improving Spyglass in the future in a number of ways. First, we plan on addressing file security by leveraging hierarchical partitioning to help eliminate partitions that the user does not have access to from the search space. Second, we are exploring new interface and query language designs that allow users to ask complex queries (*e.g.,* "back-in-time" queries) while remaining easy to use. Third, we propose fully distributing Spyglass across a cluster by allowing partitions to be replicated and migrated across machines. Fourth, we will explore how partitioning can be improved by using other metadata attributes to partition the index. Finally, we are looking at how Spyglass can be used as the main metadata store for a storage system, eliminating many of the space and performance overheads incurred when used in addition to the storage system's metadata store.

Acknowledgments

References

[1] D. J. Abadi, S. R. Madden, and N. Hachem. Column-Stores vs. Row-Stores: How different are they really? In *SIGMOD 2008*.

[2] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. In *FAST 2007*.

[3] S. Ames, C. Maltzahn, and E. L. Miller. QUASAR: Interaction with file systems using a query and naming language. Technical Report UCSC-SSRC-08-03, University of California, Santa Cruz, September 2008.

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

[5] S. M. Beitzel, E. C. Jensen, A. Chowdhury, D. Grossman, and O. Frieder. Hourly analysis of a very large topical categorized web query log. In *SIGIR 2004*.

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

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

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

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

[10] S. Buttcher and C. L. Clarke. A security model for full-text file system search in multi-user environments. In *FAST 2004*.

[11] J. R. Douceur and W. J. Bolosky. A large-scale study of file-system contents. In *SIGMETRICS 1999*.

[12] Enterprise Strategy Groups. ESG Research Report: storage resource management market on the launch pad, 2007.

[13] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM ToIS*, 2(4), 1984.

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

[15] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit groupings: Exploiting disk bandwidth for small files. In *USENIX 1997*.

[16] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole, Jr. Semantic file systems. In *SOSP 1991*.

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

[18] B. Gopal and U. Manber. Integrating content-based access mechanisms with hierarchical file systems. In *OSDI 1999*.

[19] D. Hitz, J. Lau, and M. Malcom. File system design for an NFS file server appliance. In *USENIX Winter 1994*.

[20] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *FAST 2004*.

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

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

[23] S. Khoshafian, G. Copeland, T. Jagodits, H. Boral, and P. Valduriez. A query processing strategy for the decomposed storage model. In *ICDE 1987*.

[24] N. Lester, A. Moffat, and J. Zobel. Fast on-line index construction by geometric partitioning. In *CIKM 2005*.

[25] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX 2008*.

[26] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller. High-performance metadata indexing and search in petascale data storage systems. *Journal of Physics: Conference Series*, 125, 2008.

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

[28] C. A. Lynch. Selectivity estimation and query optimization in large databases with highly skewed distribution of column values. In *VLDB 1988*.

[29] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *USENIX Winter 1994*.

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

[31] S. Mitra, M. Winslett, and W. W. Hsu. Query-based partitioning of documents and indexes for information lifecycle management. In *SIGMOD 2008*.

[32] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *USENIX 2006*.

[33] M. A. Olson. The design and implementation of the Inversion file system. In *USENIX Winter 1993*.

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

[35] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, Feb. 1988.

[36] Y. Padioleau and O. Ridoux. A logic file system. In *USENIX 2003*.

[37] Private Customers. On the efficiency of modern metadata search appliances, 2008.

[38] J. T. Robinson. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *SIGMOD 1981*.

[39] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *SOSP 1999*.

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

[41] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *FAST 2003*.

[42] M. Stonebraker and U. Cetintemel. "One Size Fits All": An idea whose time has come and gone. In *ICDE 2005)*.

[43] M. Stonebraker, S. Madden, D. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB 2007*.

[44] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI 2006*.

[45] K. Wu, E. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM ToDS*, 31(1), 2006.