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

Technical Report UCSC-SSRC-08-01
May 2008

Andrew W. Leung   Minglong Shao   Tim Bisson
Shankar Pasupathy   Ethan L. Miller

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

Minglong Shao, Tim Bisson, and Shankar Pasupathy are employees of NetApp, Inc.

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

Andrew W. Leung* Minglong Shao† Tim Bisson† Shankar Pasupathy† Ethan L. Miller*
*University of California, Santa Cruz
†NetApp Inc.

Abstract

As storage systems reach the petabyte scale, it has become increasingly difficult for users and storage administrators to understand and manage their data. File metadata, such as inode and extended attributes are a valuable source of information that can aid in locating and identifying files, and can also facilitate administrative tasks, such as storage provisioning and recovery from backups. Unfortunately, most storage systems have no way to quickly and easily search file metadata at large scale.

To address these issues, we developed Spyglass, a indexing system that efficiently gathers, indexes and queries file metadata in large-scale storage systems. Our analysis of file metadata from real-world workloads showed that metadata has spatial locality in the storage namespace and that the distribution of metadata is highly skewed. Based on these findings, we designed Spyglass to use index partitioning and signature files to quickly prune the file search space. We also developed techniques to efficiently handle index versioning, facilitating both fast update and queries across historical indexes. Experiments on systems with up to 300 million files show that the Spyglass prototype is as much as several thousand times faster than current database solutions while requiring only a fraction of the space.

1 Introduction

Modern storage systems are approaching the point where they are storing billions of files in petabytes of storage [34]. Organizing and managing this data has become a daunting task for both users and storage administrators for several reasons. Users need to find files with particular characteristics in the vast sea of data, and administrators need to understand the nature of the data being stored to more effectively manage the storage. Both tasks require the ability to efficiently answer questions about the properties of the data being stored; thus, fast, scalable searches over file metadata benefits both users and administrators.

File metadata, such as inode fields (file size, owner, modification time, *etc.*) and extended attributes, contains important information that can help in addressing these data management challenges. For example, a user may wish to find their recently modified documents or files in their home directory that should be deleted. Providing these type of file metadata queries to users can help reduce their time spent browsing and managing files. Likewise, a storage administrator may wish to find which system configuration files were recently changed or the users whose home directories have been growing the fastest to better inform their management decisions. Moreover, queries can be refined using additional metadata, such as extended attributes or a file system path to localize results to a part of the file system.

A fast, scalable metadata search system is critical for making such information easily accessible. Previous research on file search has either primarily focused on content search [5, 11, 13, 15, 18, 27, 30], which cannot address many of these queries, or relied on relational database management systems (DBMSs) to organize and index file metadata [2, 20, 21, 24]. However, through analysis of metadata from real-world workloads we show that two metadata characteristics, *spatial locality* in the file system namespace and highly *skewed distributions* of metadata values, make DBMSs an inefficient solution. This limits their ability to address the challenges in large-scale storage systems and supports the notion that existing DBMSs are not a "one size fits all" solution [6, 32]. Thus, given the need for metadata search, it is important to have a design that can achieve the scalability and performance needed to address metadata management in large-scale storage systems.

To address the shortcomings of existing systems, we developed Spyglass, a fast, scalable metadata search system designed for large-scale storage systems. Spyglass improves metadata query performance through the use of several new search and indexing techniques that exploit metadata properties. First, Spyglass uses a novel partitioning scheme that exploits the clustering of metadata values within the file system hierarchy. Second, we use signature files [8] to quickly prune the set of partitions we must search, resulting in faster searches with fewer disk accesses. Third, we utilize K-D trees [3] to provide fast search over our partitioned index. Finally, we use a new method of index versioning that enables fast time-traveling queries across multiple metadata versions. Evaluation of our prototype shows query performance improvements of up to three orders of magnitude while requiring about 10% of the space, compared to a DBMS-based approach. Additionally, Spyglass exhibits scalable performance as the size of the file system increases.

The remainder of the paper is organized as follows. We first present an extended motivation for metadata search in Section 2. In Section 3 we survey metadata characteristics and discuss why these characteristics make DBMSs an ill-suited solution. We then discuss the design and implementation of Spyglass in Section 4. We present an experimental evaluation of Spyglass in Section 5. Related and future work are discussed in Sections 6 and 7, respectively. We summarize and conclude in Section 8.

2 Motivation

In order to design a scalable metadata search system that can address user and administrator queries, it is important to understand existing storage management problems, and how searching metadata can solve them. This section discusses several use cases for metadata search, and describes common query characteristics that a metadata search system might be able to exploit. Some of these use cases are obtained through a survey of IT departments, while others are personal experiences of the authors and other users. Thus, we believe these examples are good representations of questions facing individual users and storage administrators. It is important to note, however, that these queries are not equally common, nor have the same performance requirements. For example, user queries to find their document files are likely far more common than administrator queries about the long-term growth of the storage system. Likewise, common user queries require fast performance for usability, while administrator queries have slightly more relaxed requirements, though must still be able to quickly query large sets of files.

Metadata queries can be characterized by several features, including the locality in the file system, the locality of reference, the need for metadata history, the number of query predicates (metadata attributes in a query), and the number of results returned (or its selectivity). We expect most query in a large-scale storage system will be of two classes; either, summary queries ("how much space is user *X* consuming") or queries that return a relatively short list of files. This is because queries that return very long lists of results provide no focused information, decreasing their usefulness.

Some of the most common queries are likely to be user queries to find files with particular characteristics. These queries may search the entire file system or just within a sub-tree or directory. For example, a number of us already frequently use Apple's Spotlight [2] for this very purpose on our desktops. These queries contain multiple attributes, such as modification date, file type, and owner, to produce shorter lists of results. This is because, as we will later show, any attribute alone produces too many results; however, their *intersections* are much smaller and often confine the search space to a small set of directories. Similarly, queries are often localized to sub-trees in the file system. This allows users to limit and reason about query results because queries are often not looking for files anywhere in the file system, but rather within a more specific location.

Administrative queries, while less frequent than user queries, are equally important as they aid administrator's management decisions. Here again, query results can be confined to a few sub-trees. For example, an administrator might ask about which system configuration files were recently modified or deleted. Also, both user and administrator queries have locality of reference. That is, they frequently find and search for files in only a few locations in the file system, such as a home directory or project working directory. This is because important data tends to be clustered in relatively small sub-trees.

Administrative queries often ask about summary or aggregation information. These queries allow general information about the data in the storage system to be extracted. Also, both administrator and users greatly benefit from being able to query about the past versions and the history of metadata. This can used for extracting trends over time or tracking how the storage state changes.

Our approach does not currently consider content-based queries, such as those provided by Google Desktop [12]. While content-based queries are an important class of metadata query, Spyglass does not currently handle such queries; we plan to address this area in future work.

3 Metadata Characteristics

This section discusses the characteristics of storage system metadata that make the "one size fits all" solution using a general-purpose DBMS [32] inadequate for building a high-performance metadata search system.

We would like to first define the terminology used in this paper before discussing the characteristics. *Storage system metadata*, or simply *metadata*, is a general term referring to the information describing objects, *e.g.*,files, stored in a storage system. It includes the *inode* structure used in most file systems and any extended tags added by applications or users. This paper focuses on the inode structure, although our solutions can be extended to address other types of metadata. The term *attribute* refers

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 1: ***Attributes Used.*** *They are the fields in the "inode" structure. We extract* ext *from* path*.*

Data set	Description	# of files	Server 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.*** *The small server capacity of the Eng trace is because the majority of the trace is small source code files: 99% of files are less than 1 KB.*

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

Table 3: ***Locality Ratios of the 32 most frequently occurring values.*** *All Locality Ratios are well below 1%, which means files with these attribute values are in less than 1% of directories. In other words, more than 99% of directories can be pruned from the search space.*

to the specific information of metadata, such as file size, modification time, and owner of a file. *Attribute value*, or simply *value*, refers to a value of a specific attribute. For example, "5 KB" is an attribute value of the file size attribute and `root` is a value of the file owner attribute. Attributes usually have a large set of possible values. Table 1 lists the attributes used in this paper.

The metadata used in this paper was gathered from traces taken on three network file servers—Web, Eng, and Home—in a NetApp data center; characteristics of these file servers are shown in Table 2.

Although the three traces represent different workloads, their attribute values show the same characteristics: high spatial locality in the hierarchical file system namespace and a highly-skewed distribution. The following two subsections explain these properties and their implications for the design and performance of Spyglass.

3.1 Spatial Locality of Attribute Values

The most interesting characteristic of storage system metadata is the *spatial locality* of attribute values in the hierarchical file system namespace. We make a critical observation that attribute values tend to be clustered under a few file system sub-trees. For example, files with the ext value `html` are likely to reside under directories related to web pages, and files with the owner value `tom` tend to reside in the sub-tree rooted at `/home/tom`. This locality is hardly surprising, since the hierarchical structure of file systems is used by users to classify and manage files.

We measure the spatial locality of a metadata attribute value by its *Locality Ratio*. The Locality Ratio of an attribute value is defined as the percentage of directories that contain files with that value (referred to as *target files*), compared to the number of all directories in the storage system. A directory is considered to contain a target file if any of its sub-directories contains a target file. Using this metric, an attribute value has good spatial locality if the corresponding Locality Ratio is low, meaning the target files are clustered in a few directories. Spatial locality is important for search performance, since it allows us to prune the search space to those directories that contain target files.

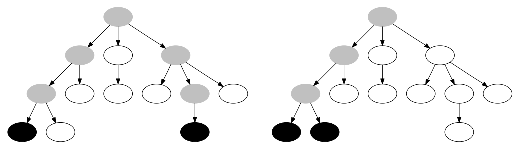
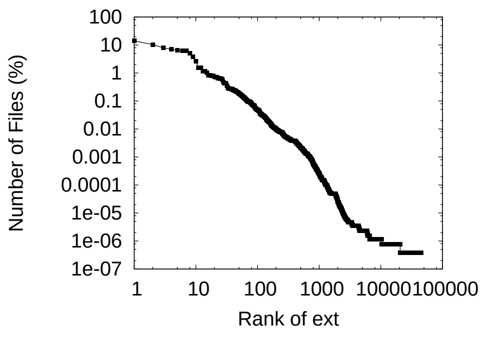

(a) **Locality Ratio=54%** (b) **Locality Ratio=38%**

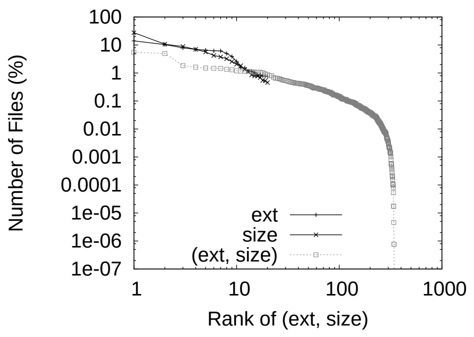
Figure 1: ***Examples of Locality Ratio.*** *The Locality Ratio of* ext *value =* `html` *is 54% (= $7/13$) in the first tree and 38% (= $5/13$) in the second tree. Therefore, the value of* `html` *has better spatial locality in the second tree than in the first one which conforms the conclusion by looking at the trees.*

Figure 1 shows a simple example of the Locality Ratio. Suppose we want to compute the Locality Ratio of html, a value of ext, for two simple file system trees. Each node in the tree graphs represents a directory. Black nodes and gray nodes are the directories that directly and indirectly, respectively, contain `html` files. It is easy to see that the Locality Ratio is a good indication of spatial locality because it correctly reflects the fact that the second tree has better spatial locality than the first one (38% < 54%). Moreover, a file system tree with better spatial locality will more quickly allow a system to prune directories that *cannot* contain a file that matches a query.

We calculated the Locality Ratios of the 32 most frequently occurring values of different attributes in Web, Eng, and Home, summarizing the results in Table 3. This table lists Locality Ratios of 4 attributes: ext, size, owner, and ctime. Other attributes have similar Locality Ratios which are omitted to save space. In all cases, attribute values show very good locality in the file system



(a)



(b)

Figure 2: ***Attribute Value Distribution Examples.*** *A rank of 1 represents the attribute value with the highest file count.*

namespace ($\ll 0.01$); thus, more than 99% of directories can be pruned from the search space.

Unfortunately, current metadata search solutions are either namespace-oblivious, such as the DBMS solutions treating path names as normal character strings, or unaware of spatial locality, such as brute-forth search in `find`. In contrast, the search algorithm and data structures in Spyglass exploit spatial locality to achieve better performance, as demonstrated by our experimental results in Section 5; Section 4 describes the Spyglass search algorithm in detail.

3.2 Skewed Distribution of Attribute Values

Another prominent characteristic of storage system metadata is the highly skewed data distribution for almost all attribute values and combinations of attributes. Figure 2 shows the distributions for ext and the combination of (ext, size) from the Home trace.

The figures are plotted as follows. Taking Figure 2(a) as an example, we count the number of files for each ext value (such as `html`, `doc`, and `pdf`) and rank all ext values based on their file counts with rank 1 being the ext value that has the most number of matching files. We then plot the ranks of ext values (the X axis) and their corresponding file counts in percentage (the Y axis) using a log-log scale. Figure 2(b) is plotted in a similar way using values of the two-attribute pair (ext, size).

Figure 2(a) shows that 80.0% of files have one of 20 popular values of ext while the remaining 20.0% of files account for over 40000 other file extensions. Overall, the distribution curve is similar to the *power law* distribution [29]. This observation holds true for other attributes across all traces we examined. We do not graph these due to lack of space.

We next generated a Cartesian product of the top 20 values from ext and size, yielding $20 \times 20 = 400$ different pairs. We can see from the file counts (in percentage) shown in Figure 2(b) that the file counts of these pairs are significantly smaller than the file counts for the corresponding single attribute; most are at least an order of magnitude smaller. For attribute combinations that involve more than two attributes, their file counts are even smaller.

The above two figures show two observations on storage system metadata: searching for popular values of a single attribute results in a large set of matching files; but searching for combinations of multiple popular single-attribute values often results in a very small set of matching files. Therefore, indexes that can *simultaneously* search on multiple attributes to obtain the final matches directly are the best solutions for Spyglass, where the majority of queries on metadata are multi-attribute queries, as discussed in Section 2.

Existing solutions in DBMS using single-attribute indexes, such as index ANDing [7] or composite indexes [25] cannot avoid unnecessary processing on unwanted intermediate results, making them inappropriate for Spyglass. Rather, multidimensional access methods [9], also known as multidimensional indexes, offer better solutions for Spyglass. Among a variety of multidimensional access methods, the design of Spyglass uses K-D trees [3], a popular multidimensional access method, to improve the performance of multi-attribute search. Section 4 explains how Spyglass adopts K-D trees to search storage system metadata and how it balances the trade-offs of K-D trees.

4 Design and Implementation

The goal of Spyglass is to aid data management by providing a scalable, search-able repository of all file metadata in the storage system. Our design was guided by several principles: (1) The index should be sensitive to the file system's hierarchy. The hierarchy already defines how users organize and group files, and contains information about how files are accessed and used. The index should exploit this information. (2) Fast query execution is more important than strict consistency. Most queries can be ad-

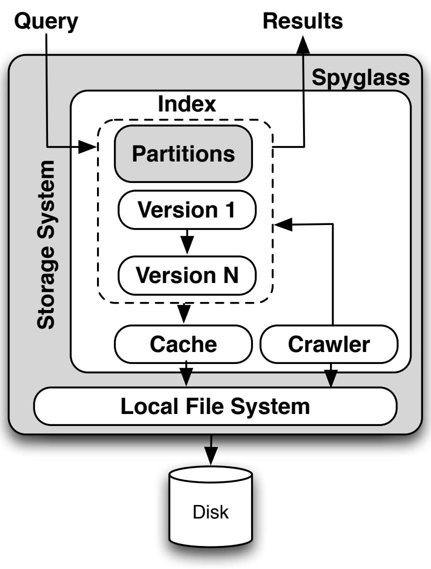


Figure 3: ***Spyglass Overview.*** *Spyglass resides within the storage system. The crawler extracts file metadata, which gets stored in the index. The index consists of a number of partitions and versions.*

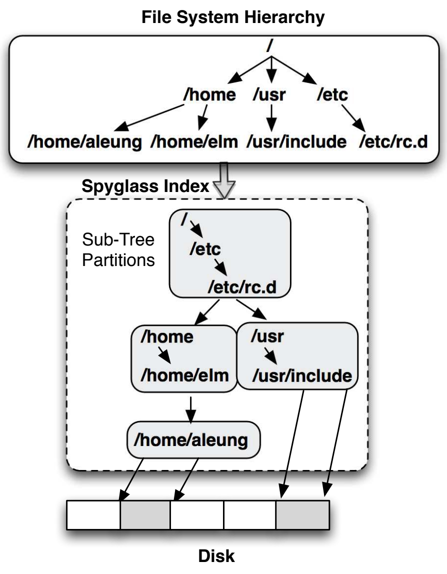


Figure 4: ***Hierarchical Partitioning Example.*** *Sub-trees partitions, shown in gray, index different file system sub-trees. The Spyglass index is simply a tree of sub-tree partitions.*

equately satisfied even with slightly stale data, while performance is critical to usability. (3) File metadata history should be maintained because it facilitates queries regarding usage and storage trends. (4) Dedicated hardware resources should not be required as they may become prohibitively expensive in large-scale systems. Instead, Spyglass should be able to reside within the storage system.

In addition to these principles, we have chosen to focus on several types of queries we believe are the most important and most likely. 1) Multi-dimensional queries with more than one query predicate. Specifying queries with multiple predicates is an effective way of refining search results. In a large-scale storage system, single predicate queries often return too many results to be of use. 2) Queries localized to a sub-tree or directory. Localizing queries allows users to control which files to search. In large-scale storage systems, searching the entire namespace is often not needed as users can reason about the location of the files they care about. 3) Time-traveling queries. Querying across multiple metadata versions allows users to understand how storage is used and how it changes. This information can improve how users organize their data.

Spyglass consists of two major components: the Spyglass index which stores metadata and serves queries and a crawler that extracts metadata from the storage system. Figure 3 provides a high-level view of Spyglass. The Spyglass index design utilizes two key concepts: *hierarchical partitioning* and *partition versioning*. Hierarchical partitioning decomposes the index into separate partitions based on the storage namespace hierarchy. This allows the index to be managed and searched at the granularity of sub-trees, which is critical to providing scalability as the system grows. Partition versioning manages index updates and versions. Index updates are batched and applied to each partition as new incremental versions. Versioning updates enable users to query over past versions and simplify index update semantics. Throughout this section we discuss the motivations behind these concepts and how they are applied.

4.1 Hierarchical Partitioning

The Spyglass index is partitioned into a number of separate indexes based on the file system's namespace hierarchy. The concept of hierarchical partitioning is a diversion from the traditionally row, and more recently column [31], physical designs of DBMSs. Rather than store records physically adjacently on-disk using their row or column order like DBMSs, Spyglass stores records adjacently on-disk that are hierarchically nearby in the namespace. Hierarchical partitioning is illustrated in Figure 4, where sub-trees are mapped to separate partitions, shown in gray. Each partition is stored sequentially on-disk. The motivation behind this design is that queries can often be satisfied with only a small fraction of the hierarchy. For example, searching for user `aleung`'s presentation files likely does not require searching all sub-trees. Likewise, localizing queries to a sub-tree reduces the search space. As a result, only a small subset of the hierarchy often needs to be retrieved from disk. By clustering the hierarchy on-disk and allowing only portions of the hierarchy to be read at a time, queries can often be satisfied with only a few small sequential reads, even in large-scale systems.

In addition, hierarchical partitioning utilizes locality in file metadata and access patterns. The spatial locality analysis in Section 3.1 shows that files nearby in the hierarchy are more likely to share metadata values than those farther apart. For example, most or all files in a personal directory may share a common owner, or files in a directory may share a common modification time. This is also true for extended attributes, where only a small related set of files tend to share attribute keys. As a result, fetching a partition of the hierarchy from disk will often fetch a number of qualified records. Likewise, not fetching records from all parts of the hierarchy can often reduce the number of unqualified records fetched. File access patterns also exhibit locality. More precisely, not all directories and sub-trees are equally popular [1, 19]. Often only a small fraction of directories, relative to the entire hierarchy, are frequently accessed. This implies that only a fraction of hierarchical partitions may be frequently queried, which can be stored in-memory with infrequently queried partitions stored on-disk.

We refer to each hierarchical partition as a *sub-tree partition*. A sub-tree partition manages metadata for one or more of the file system's sub-trees. In Figure 4 we see different sub-trees map to different partitions. Our current prototype uses a simple greedy algorithm to do this mapping. The Spyglass index is simply a tree of sub-tree partitions. The tree's parent-child relationships are based on where the sub-tree appears in the hierarchy. A sub-tree partition has two components: a *partition index* and *partition metadata*. The partition index stores and serves queries for metadata in its assigned sub-trees. Partition metadata is used to determine if a partition is relevant to a query, aid aggregation queries, and support partition versioning.

The entire Spyglass index is stored on-disk. However, a copy is kept in-memory, with the exception of the partition indexes. The Spyglass index tree and partition metadata are small, however the partition indexes, which stores the metadata, are too large to all fit in-memory. Instead, a *partition cache* is used to manage the paging in and out of partition indexes from memory. The partition cache pages entire partition indexes to and from memory. The motivation is that if a file must be read for a query, it is likely that other files nearby in the hierarchy also need to be read; paging entire partition indexes allows these files to be fetched in a small sequential read. This concept is analogous to file system embedded inodes [10]. Embedded inodes store inodes adjacent to their parent directory on-disk. This allows the directory and its inodes to be fetched in a small, sequential read under the assumption that an access to one directory or inode will likely access other inodes in the directory.

The partition cache uses a simple LRU algorithm to manage memory. Similar to that of file system caches, queries have locality of reference and a partition index queried once is more likely to be queried again. In the common-case, only a small set of partitions, corresponding to popular sub-trees, are frequently searched. An LRU algorithm keeps these popular partitions in-memory, while most reside on-disk. As a result, querying commonly accessed sub-trees will produce no disk accesses and be very fast.

4.2 The Partition Index

The goal of the partition index is to quickly satisfy requests for all metadata in a sub-tree partition. To do this we use a K-D tree [3]. A K-D tree is a k-dimensional binary tree that provides logarithmic point, range, and nearest-neighbor search over a k dimensional space. It traverses the tree by alternating the dimension $(1 \ldots k)$ used to pivot at each level. Each metadata attribute is a unique dimension in the K-D tree. A K-D tree is used because it provides fast, multi-dimensional search over all metadata in the partition. Alternative multi-dimensional structures, such as R-trees [14], Grid Files [23], and K-D-B-trees [26] either perform poorly for non-uniformly distributed data or are disk-based structures. Also, space overhead is minimal because beyond the file metadata, only the tree pointers need to be stored.

A K-D tree is poor for frequently changing data because it can perform poorly when unbalanced. This means frequent metadata updates can degrade performance. However, partition versioning, which manages index updates, treats updates as immutable versions. Therefore, a K-D tree is never updated in-place and will not become unbalanced. We discuss partition versioning further in Section 4.4.

4.3 Partition Metadata

Each sub-tree partition also contains metadata about the files and sub-trees it indexes. This includes the names of the indexed sub-trees, summary statistics, version information, and *signature files* [8]. Summary statistics aid aggregation and trend queries. Statistics, such as minimum, maximum, and average values are computed for each metadata attribute when the sub-tree partition is updated. By pre-calculating statistics, aggregation and trend queries can be satisfied without needing to read or process information from the partition index. Each partition also maintains a *version vector*, which is a vector of different partition index versions. We elaborate on version vectors in Section 4.4.

Each sub-tree partition contains a signature file for each indexed metadata attribute. Signature files, or just signatures, are compact summaries of a partition index's contents. Signatures are used during query execution to determine if a partition index needs to be searched by determining if it contains any files matching the query predicates. Note, signatures only *tests* for the existence of matched files. The partition index must be queried to retrieve its information.

A signature is a bit-array with an associated hashing function. All bits in the signature are initially set to zero. A common example is a Bloom Filter [4]. When a metadata value is inserted, it is hashed to a bit-position, modulo the size of the signature, which is then set to one. A query only searches a partition if signature bits for all predicates in the query are set to one. This allows Spyglass to quickly test which partitions are needed for a query and which are not. However, a signature can only determine if a partition index does *not* have any records matching the query predicates. This is due to hashing collisions which can cause false-positives. False-positives do not affect the accuracy of results because a partition with matching results will never be skipped. However, false-positives can degrade performance by causing partitions with no matching results to be searched. Spyglass controls false-positives two ways. First, each partition index is kept relatively small, keeping the chances of collision low. Second, metadata attributes can use specialized hashing functions that have more control over false-positive occurrences. For example, a signature of file sizes may wish to assign each bit to a size range rather than a unique size value. This can allow false-positives to be clustered for the frequently occurring size ranges, reducing false-positives for less popular ranges.

4.4 Partition Versioning

Spyglass receives index updates in batches and treats each batch as a new version. The motivation is two fold. First, we wish to support time-traveling queries as they enable useful data management queries. Second, we wish to simplify index update semantics. Designing for frequent, in-place updates greatly complicates design as locking and synchronization must be considered. Also it degrades query performance as queries and updates contend for shared data structures and cache space. This motivation follows from our design principle that index consistency is, in general, less important than query performance as most queries do not require strict consistency. As a result, Spyglass trades-off index consistency for scalable time-traveling queries and simple update semantics.

4.4.1 Creating Versions

Updates are applied in batches of incremental metadata changes and each update represents a new index version.

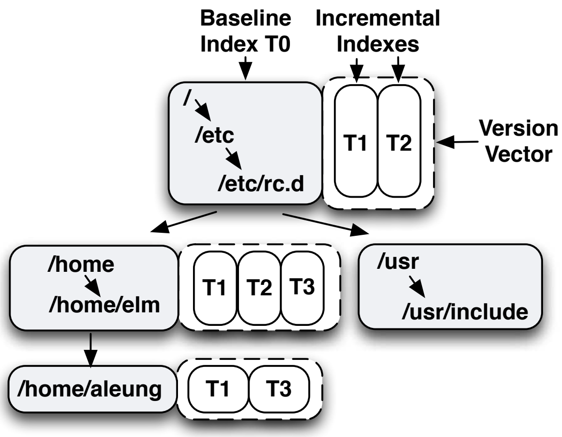


Figure 5: ***Versioning Partitioning Example.*** *Each sub-tree partition manages its own versions. A baseline index is a normal partition index from some initial time T_0. Each incremental index contains the changes required to roll query result forward to a new version. Each sub-tree partition manages its version in a version vector.*

We discuss how incremental changes are collected later in Section 4.5. Each sub-tree partition manages new versions for its assigned sub-trees, meaning individual partitions are versioned rather than the Spyglass index as a whole. This is shown in Figure 5. A versioned sub-tree partition contains two components: a *baseline index* and *incremental indexes*. A baseline index is a normal sub-tree partition index and represents the state of the storage system at time, T_0. An incremental index is an index of metadata *changes* between two points in time, T_{n-1} and T_n. An incremental index contains the information needed to roll query results from T_{n-1} forward to T_n. These changes include metadata creations, deletions, and modifications. By storing just incremental changes, partition versioning has minimal space overhead.

Partition versioning begins with a baseline index, as shown in gray and labeled T_0, in Figure 5. When a batch of metadata changes are received at T_1 they are used to build incremental indexes. Each sub-tree partition manages its incremental indexes using a version vector, which is a vector of incremental indexes, each representing a different version. We see in Figure 5 that each partition's vector can be a different length because partitions are likely not updated at the same rate. The partition cache also manages incremental indexes and pages them in and out with the baseline partition index. As a result, partition versioning adds an overhead to page-in a partition as all incremental indexes must also be read. The motivation behind this is that due to locality of reference, queries will often hit in the partition index cache. Thus, in the common-case, versioning introduces a small overhead because no additional disk accesses are required.

To retrieve query results from any index version, T_n, the results from the baseline index, T_0, and *all* changes between T_0 and T_n are needed. For example, in Figure 5 querying the sub-tree `/home/aleung` requires the baseline index and results from incremental indexes T_1 and T_3. The changes from T_1 and T_3 modify the results (add or remove results) from T_0 to produce query results that reflect the state of the storage system at T_3. Because each incremental index only contains changes for a partition, retrieving and applying changes is often very fast. This means that the partition versioning overhead is dominated by the number of partitions to page-in from disk in order to satisfy a query.

The goal of an incremental index is to quickly retrieve the metadata needed in order to roll the baseline query results forward to a more recent version. To do this we again use a K-D tree. Each K-D tree indexes metadata changes. Metadata changes include the type of change (create, delete, or modify) and the changed metadata. Changes that create metadata include the newly created metadata, changes that delete metadata include the deleted metadata, and changes that modify metadata include the old and new metadata. This is because a modification can cause a new metadata version to match a query, as well as, cause old metadata versions to no longer match a query. Thus, we must be able to add the new results that match the query and remove the old results that no longer match from the results lists.

4.4.2 Managing Versions

While maintaining many incremental versions can facilitate useful queries, they also add space and time overhead. Over time it becomes less useful to keep older version at a fine time granularity. To reduce the overhead for older versions, Spyglass uses *version collapsing* and *version checkpointing*. Version collapsing merges incremental indexes with a baseline index, reducing overhead by removing the incremental index at the cost of version granularity. When collapsed, an index becomes accurate to the time of the last collapsed incremental index. Version checkpointing allows a collapsed index to be saved to disk and represents a landmark versions of the index. A landmark version is a full Spyglass index that is retained, as it represents some significant point-in-time.

We describe the use of collapsing and checkpointing using an example. Suppose that the Spyglass index is updated hourly, creating a new incremental version of the index. Time-travel can be performed at hourly granularity. At the end of the day, incremental versions can be collapsed into the baseline index. This reduces time and space overhead, however we can no longer travel hour-by-hour over the last day. Also, at the end of each day, each collapsed index is checkpointed. These checkpoints represent storage system state at the end of each day. At the end of the week, all but the latest daily checkpoint are deleted. Likewise, at the end of the month, all but the latest weekly checkpoint are deleted. This results in different time-scales maintaining different version granularity. Over the past day any hour can be searched. Over the past week any day can be searched, over the past month any week can be searched, and so on. Managing index versions this way allows time and space to be traded-off for the required time-traveling capabilities.

4.5 Collecting Metadata Changes

To collect batches of metadata changes Spyglass takes advantage of snapshot technology in the WAFL [16] file system on which we designed our prototype. Our approach allows the difference between two file system snapshots to be quickly calculated. This provides a fast method for generating the batches of update Spyglass needs. It should be noted that Spyglass does not depend on this approach. Any method for collecting metadata changes will suffice. However, alternative solutions presented us with a number of challenges. Periodically walking the file system tree is a time consuming process. Also, buffering file system event notifications to generate batches of changes can require large amounts buffer space.

Given two file system snapshots, we quickly calculate metadata differences between them using WAFL's *inode file*, a file containing all inodes in a snapshot. When a snapshot is created in WAFL, it copies the inode file using a copy-on-write mechanism. As a result, we can simply read each snapshot's inode file, and compare them to generate the metadata changes between two snapshots. Snapshot-based differencing is very fast because it only needs to compare the inodes that have changed between the two snapshots (due to copy-on-write). The output is a log of all added, deleted, or modified metadata. If only one snapshot is used, a log equivalent to a crawl of the entire file system is produced.

5 Experimental Evaluation

In this section, we evaluate our Spyglass prototype. The goal of the evaluation is to understand performance and scalability properties and how they compare to existing DBMS solutions. Overall, Spyglass achieves fast query execution, hundreds of milliseconds for common queries, even as the number of files increases. Spyglass also consumes less space and has better update performance than DBMSs. The versioning mechanism of Spyglass is efficient which incurs little overhead for most queries.

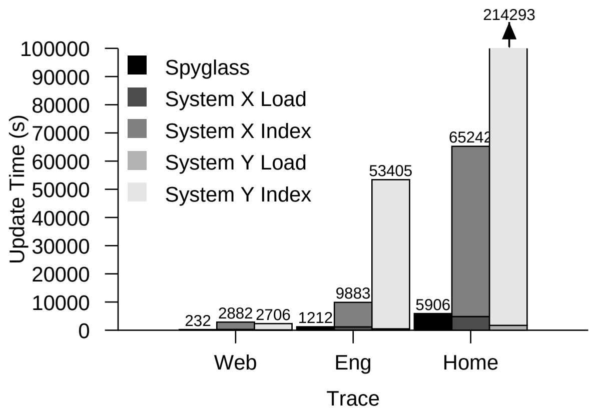


Figure 6: ***Comparison of Update Performance.*** *All times are reported in seconds. It measures the time to update Spyglass with no incremental indexes and the time to load the table and build indexes in the DBMSs. Spyglass is 8 to 44 times faster than the DBMSs.*

5.1 Experimental Setup

All experiments were run on a Dual core AMD Opteron machine with 8 GB of RAM running Ubuntu Linux 7.10. Related files and data sets are stored on an NFS partition, mounting a high-end NetApp controller.

All experiments use the same metadata traces as described in Table 2. For Web and Eng, we also collect several days of incremental snapshot metadata. Each contains daily changes of all metadata.

We compare Spyglass to two popular relational DBMSs, anonymously referred to as System X and System Y. For both DBMSs, we use an index-based physical design, which consists of a base relation with all attributes in Table 1. Each attribute has a B+-tree index built on top of it. Spyglass uses the same attributes when building its K-D trees. We use this design, as opposed to vertical partitioning or composite indexes, because it is easy to implement and we believe is a likely design choice for a metadata DBMS.

Internal cache sizes are set to 128 MB, 512 MB, and 2.5 GB, for the Web, Eng, and Home traces, respectively, in all three systems. This amounts to about 1 MB for every 125,000 files. Spyglass also uses a soft limit of 100,000 files per sub-tree partition index. There are no limits on the size of an incremental index.

5.2 Microbenchmarks Evaluation

Our microbenchmarks evaluate update and metadata collection performance, space overhead, Spyglass index locality, and selectivity sensitivity.

5.2.1 Update Performance

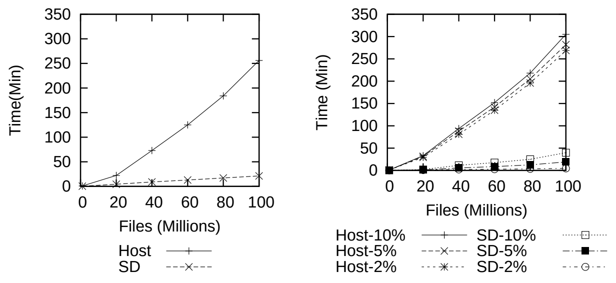
We compare the performance of updating baseline indexes of all metadata traces in Spyglass to the performance of bulk loading and index building in the two DBMSs. We do not look at incremental index update performance as the DBMSs have no versioning, making an accurate comparison difficult.

Figure 6 shows that Spyglass is between 8x and 44x faster than System X and System Y. Spyglass indexes all attributes of each metadata entry once and usually writes to disk in relatively large sequential streams. In contrast, each DBMS indexes each attribute separately, in addition to loading the table. Spyglass is still faster even if we compare only the indexing time in the DBMSs with the total update time in Spyglass. To put it in perspective, Spyglass updates the 300 million Home trace files in one and a half hours, while the DBMSs take 18 hours and 2 and a half days, respectively. Last but not least, Spyglass update performance shows a linear scalability with regard to trace sizes. The performance difference between the DBMSs is due to the significant differences in how each builds indexes.

5.2.2 Metadata Collection Performance

We now show the performance of baseline and incremental crawling using our snapshot-based file system crawler, and compare it to an optimized multi-threaded crawler that walks the file system tree to compute snapshot differences, which we call the host-based crawler. Figure 7(a) shows the time to generate a baseline using both the host-based crawler and the snapshot-based crawler. A baseline crawl generates a complete list of all metadata for a given file system hierarchy. This figure shows see that the snapshot-based crawler outperforms the host-based crawler; the snapshot-based crawler can leverage the on-disk layout of file metadata by sequentially scanning the inode file and reporting each file's metadata. The host-based crawler on the other hand must traverse the file system hierarchy and then sort the metadata. The host-based crawler generates a baseline in a sorted order to facilitate incremental crawling.

An incremental crawl reports the changes between two versions (snapshots) of a file system. For the host-based crawler, an incremental crawl is generated by creating a second baseline, then sequentially scanning the two baselines to determine their differences. Figure 7(b) shows the time to generate the incremental changes between two file system versions when the relative changes to the baseline are 2%, 5%, and 10%. For instance, the plot Host-5% at 40 million files means a change of 2 million files. This figure shows that the snapshot-based crawler significantly outperforms the host-based crawler. The snapshot-based crawler is able to avoid comparing blocks making up the



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

Figure 7: ***File System Crawling Performance.*** *Compare Host-based crawler (Host) and Snapshot-based crawler(SD).*

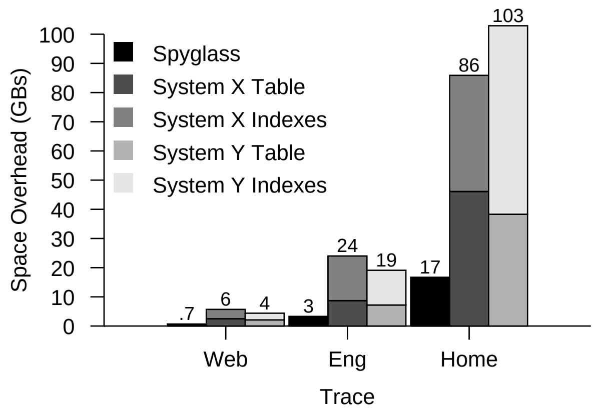


Figure 8: ***Comparison of Space Overhead.*** *All numbers are reported in gigabytes. For the DBMSs, this measures the space consumed by the table and B+-tree indexes. Spyglass requires 5x to 8x less space than either System X or System Y.*

inode file that haven't changed between two snapshots because of WAFL's copy-on-write mechanism. As a result, incremental snapshot-based crawl performance is relative to the number of changed files, not the total number of files. However with the host-based crawler, performance is relative to the total number of files because it must first generate a sorted baseline, then difference the two baselines.

5.2.3 Space Overhead

This section examines the on-disk space consumed by all three systems. For the DBMSs, this includes the table and B+-tree index space. Figure 8 shows Spyglass takes 5x to 8.5x less space than either DBMS. This is due to two main reasons. First, Spyglass indexes each metadata entry only once whereas the DBMSs require the table space plus the space for each index where each stores *N* (value, record id) pairs (minus duplicate values). The index space alone is larger than the total space in Spyglass. Second, Spyglass saves space by storing only partial pathnames because part of the path prefix is already stored in the sub-tree partition's metadata. This also explains why Spyglass consumes less total space than just the DBMS tables.

Once again, Spyglass shows very close to linear scalability across the traces. Since disk space is cheap, the ramifications of space overhead is often not the on-disk footprint, but rather the number of records that can be stored in-memory. With a smaller space overhead, a larger fraction of the index can be stored in memory, reducing disk accesses, thereby improving query performance.

5.2.4 Index Locality

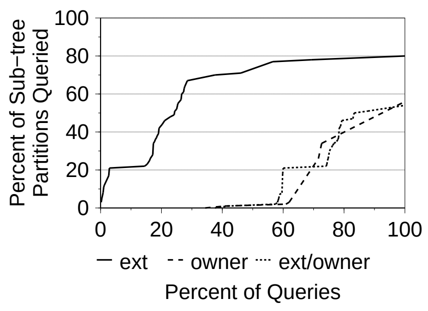
Here we measure how effectively Spyglass exploits locality in the namespace hierarchy. To do this, we generate a query log for each attribute and each two-attribute pair based on the template "select files with attribute = value" and "select files with attribute1 = value1 and attribute2 = value2", respectively. For example, the query template for ext is "select files with ext = value". Each log consists of 300 queries with values randomly selected from a full metadata trace and 200 queries with values randomly selected from the corresponding incremental trace, resulting in a total number of 500 queries. The reason of using the incremental traces is to incorporate a notion of popularity into the query logs because the incremental traces represent frequently accessed files.

Due to the space limit, we report only the results of executing the query logs for ext, owner, and (ext, owner) from the Eng trace. Recall that Spyglass uses signature files to eliminate sub-tree partitions from the search space. A partition is queried only if all signature bits corresponding to the query predicates are set to one.

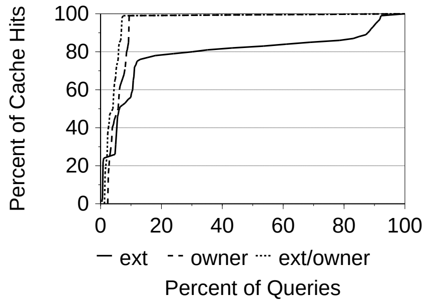
Figure 9(a) shows a cumulative distribution (CDF) of sub-tree partitions queried for each query log. We find that 50% of the queries on ext reference fewer than 75% of the sub-tree partitions, while over 50% of the (ext, owner) and owner queries reference fewer than 2% of sub-tree partitions. This is because the owner attribute is more clustered in the hierarchy than ext, which confirms the findings in Table 3. In addition, the (ext, owner) queries reference far fewer indexes than ext or owner alone. This is because the combination of the two attributes is highly clustered. Multi-attribute queries often provide better locality than single-attribute queries.

Figure 9(b) shows a CDF of the cache hit rates in our query logs. Again, we find that the ext queries have worse locality than either of the other logs. Only 22% of ext queries have a cache hit ratio of 85% or higher while 91% of the owner queries have a cache hit ratio of 99% or higher. Because ext values are more distributed throughout the hierarchy, it is less likely that a queried partition is already in the cache.

Experiments on other query logs across the three data sets have similar observations. In summary, these mea-



(a) CDF of sub-tree partition accesses.



(b) CDF of partition cache hits.

Figure 9: ***Index Locality.*** *Figure 9(a) is a cumulative distribution function (CDF) of the fraction of sub-tree partitions accessed. Figure 9(b) is a CDF of the fraction of partition cache hits. Our query logs issue single attribute queries with* ext *and* owner *and two attribute queries with both. We find that Spyglass queries only a fraction of the partitions, with few disk accesses, for* `ext` *and* `owner` *combinations queries.*

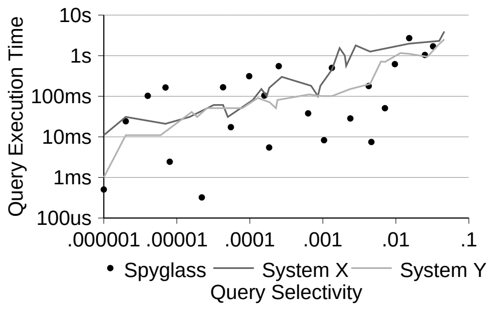


Figure 10: ***Comparison of Selectivity Impact.*** *The selectivity of queries in our query log is plotted against the execution time for that query. We find that query performance in Spyglass is much less correlated to the selectivity of the query predicates than the DBMSs, which are closely correlated with selectivity.*

surements show that Spyglass is effective at limiting the search space and disk access because it can leverage the spatial locality existing in storage metadata.

5.2.5 Selectivity Impact

In this experiment, we compare how metadata selectivity influences the performance of Spyglass and the DBMSs. We again generate query logs of ext and owner from the Web trace with varying *selectivity* (# of results / # of all records). Figure 10 plots query selectivity against query execution time. We find that the performance of System X and System Y are highly correlated with query selectivity. However, this correlation is much weaker in Spyglass, which exhibits much more variance. For example, a Spyglass query with selectivity 7×10^{-6} has a 161 ms run time while another with selectivity 8×10^{-6} has a 3 ms run time. This is because Spyglass is more sensitive to hierarchical locality and query locality than query selectivity. This is unlike DBMSs, which access records from disk based on the predicate it thinks is the most selective. The weak correlation with selectivity in Spyglass means it is less affected by the highly skewed distribution of storage metadata which makes determining selectivity difficult.

5.3 Macrobenchmark Evaluation

We now compare the performance of Spyglass with System X and System Y on a macrobenchmark generated based on three query logs that mimic real possible user queries. Each query log represents a different kind of query a user may ask. The first is a user finding the space consumed by their files of a particular type. This involves queries with owner and ext predicates, retrieving and summing file size (size). The second is a user locating files in their personal directory. This involves queries with owner, type, and path predicates. Matched results must have a prefix that matches the path predicate. File inode numbers (inumber) are returned. The third is a user locating their recently modified files. These queries involve owner, ext, path, and time range predicate. The time is a two-week range over mtime. These templates are chosen because each looks at the impact of different query types on the index.

We generate our macrobenchmark by filling each query log with attribute values randomly selected from each trace. By randomly choosing values, the frequency distribution of attribute values is maintained in the query log, meaning more frequently occurring values are more frequently queried. When randomly selecting files, we ignore files with high hard link counts because they skew hierarchy locality and are an aberration from normal metadata properties. As a result, not all query logs have the same number of queries. The total number ranges between 100 and 300 queries. All queries in a log are replayed in the order they appear in the trace file. We find 3x to 5x performance differences between System X and System Y in our experiments. We believe this is due to the differences in how each chooses query plans, resulting in the occasional use of table scans.

Figure 11 compares the total run times of each query log on each trace. For the first query log, Spyglass is between 3.5x and 18x faster than the DBMSs. This is be-

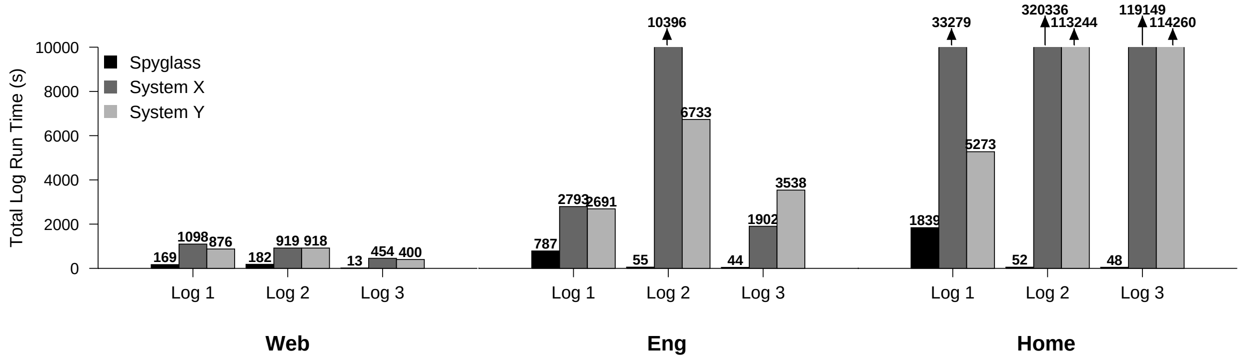
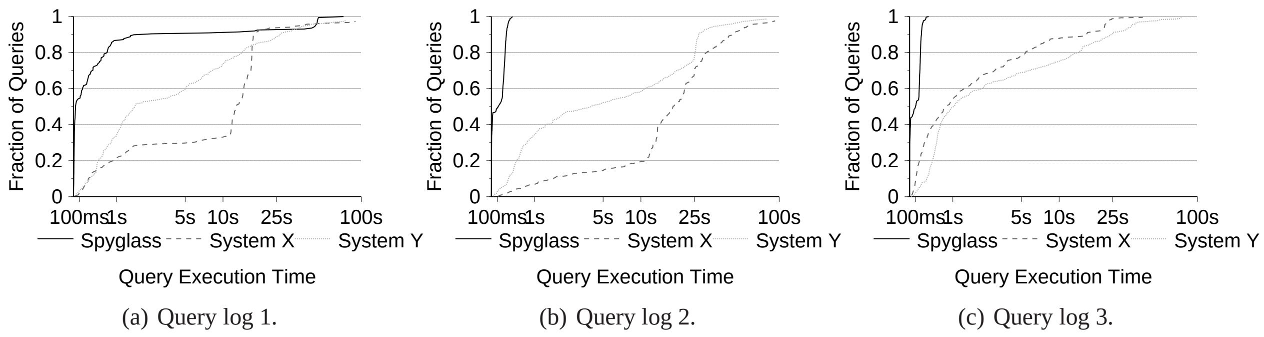


Figure 11: ***Macrobenchmark Run Times.*** *We show the time required to run each macrobenchmark query log. Each query log is labeled 1 through 3 and are clustered by trace file. Bars that extend beyond the plotting area are labeled with an arrow. We find Spyglass outperforms the DBMSs on all query logs, especially the second and third query logs. This is because the index partitioning in Spyglass can significantly narrow the search space.*

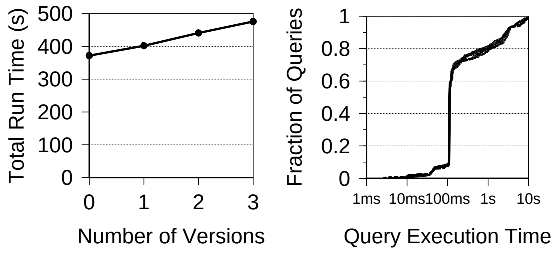


(a) Query log 1. (b) Query log 2. (c) Query log 3.

Figure 12: ***CDFs of Macrobenchmark Query Execution Times.*** *For each query log, we show a CDF of query execution times for the Eng trace. In Figures 12(b) and 12(c) all queries are extremely fast because these query logs include a file path predicate that allows Spyglass to narrow the search to a few partitions.*

cause Spyglass is usually able to narrow the search to a small number of sub-tree partitions. Figure 12(a) shows the CDF of query execution for this query log on the Eng trace. We see that 54% of Spyglass queries have an execution time less than 100 ms. This shows that the query execution hits the partition cache most of time and has very few or no disk accesses. However, we see that the curve tapers. This is because a number of queries either access many partitions that are not in the cache or access more partitions than the cache can hold.

For the second and third query logs, we find that Spyglass significantly outperforms the DBMSs: three *orders of magnitude* ($> 1000\times$) in some case. The key reason for the improvement lies in the hierarchical partitioning. The hierarchical nature of the Spyglass index allows sub-trees of the hierarchy to be quickly searched, without the need to process or traverse other locations. These query logs use path as a predicate, which allows Spyglass to only search sub-tree partitions below the path. Figures 12(b) and 12(c) demonstrates this with a CDF of query times on the Eng trace. Almost all queries finish within 100 ms. This is because the search space is often narrowed to only a few sub-tree partitions ensuring a worst-case scenario of a few sequential disk accesses if the partitions are not in cache. In summary, Spyglass exploits the locality properties of both the metadata and queries to reduce the overall search space, allowing it to scale in large-scale storage systems.



(a) Total run times. (b) CDF of run times.

Figure 13: ***Partition Versioning Performance.*** *The total run time of the 500 queries increases 10% for an additional incremental index. The overhead is caused by only a few queries.*

5.4 Partition Versioning

We now look at the performance overhead of partition versioning. We use the full baseline Web trace and its three incremental traces, which are the metadata changes in the three days following the baseline. We use this data to generate a query log using the same method discussed in Section 5.2.4. Figure 13(a) shows the query log's running time with no incremental indexes (just a baseline index) and with one, two, and three incremental indexes. We see that each incremental index adds a 10% overhead to the total running time, which scales linearly. Figure 13(b), which is a CDF of query execution time, shows that the 10% overhead is not evenly distributed amongst the queries. We see that the distribution of query execution time is very close for all curves. This is because versioning adds very little overhead when the sub-tree partition is already cached. Cache hits require only microseconds to query an incremental index. However, cache misses must read the partition index and all of the incremental indexes from disk. Figure 13(b) shows that for most queries, overhead is very low. For the few queries that require a number of disk accesses, overhead increases, which accounts for the 10% overhead in the total running time.

6 Related Work

As the amount of data in storage systems has grown, more work has focused on effectively managing it. Past research focused on semantic data search [5, 11, 13, 15, 27] and more recently, extracting and searching semantic relationships, such as context [30] and provenance [22, 28]. Using search to manage storage has also found its way into available products [2, 12, 17, 20, 21]. However, much of this work has been focused on content search. While useful, content search only provides the ability to locate files based on content keywords. As a result, it lacks many important queries offered by metadata search. Some work does address metadata search, though it is often left to general-purpose DBMS systems, which are ill-suited solutions. We believe Spyglass addresses a key component of effective data management and can be use to aid existing content search systems.

Spyglass also follows in the spirit of the database community that "one size fits all" DBMS solutions do not work [6, 32]. This paradigm argues that the best data management solutions are those designed specifically for the problem at hand, which has produced new database designs, such as H- and C-stores [31, 33]. However, data management in storage systems has largely ignored this idea. We feel Spyglass is a first step towards making data management and search primary a component of the storage system by showing performance and scalability can be achieved with specialized designs.

7 Future Work

Thus far, Spyglass has addressed scalable metadata search, however, there are a number of important data management aspects not yet addressed. Two that we plan to look at in the future are query language and security. An effective query language is important for the system's usability, however, a number of important queries, such as time-traveling or trend queries, do not map well onto existing languages, such as SQL. We believe a specialized query language, like our indexing structures, can provide significant benefit over existing tools. Security is also important for usability because it must not leak information to a user about the contents of the storage system that they are not authorized to see. However, since access control in file systems is often at the granularity of sub-trees, Spyglass can leverage hierarchical partitioning to improve the time spent performing security checks.

We view Spyglass as a first step towards enabling users and administrators access to their data beyond traditional directory browsing mechanisms. We plan to look at how to integrate scalable file content search into large-scale storage systems. We also plan to look at how information beyond a file's metadata and content, such as relationships with other files, can be integrated into the storage system.

8 Conclusions

Managing and organizing data has become much more difficult, for both storage users and administrators, as storage systems have begun storing much more data. In this paper, we argued that the ability to search file metadata has the potential to address a number of these problems. Searching file metadata allows users and administrators to quickly gather information about storage that improves how they manage data. We showed that metadata has both spatial locality and skewed distributions, limiting the performance and scalability of existing solutions that use DBMSs.

To address this issues, we developed Spyglass: a fast, scalable system for searching metadata in large-scale storage systems. Spyglass uses novel indexing techniques that partition the index based on the file system hierarchy to exploit locality of metadata values and applies signature files to quickly prune the query search space. Spyglass also includes a novel index versioning method to allow index updates and queries based on index history. An evaluation of our Spyglass prototype shows that it can outperform DBMS solutions with respect to time-space overhead, update time, and query performance, reducing query time by up to three orders of magnitude for some macrobenchmarks while only consuming 10% of the storage space compared to traditional DBMS-based metadata indexing techniques.

Acknowledgments

This work was supported in part by the Department of Energy under award DE-FC02-06ER25768 and industrial sponsors of the Storage Systems Research Center at UC Santa Cruz, including Agami Systems, Data Domain, Hewlett Packard, LSI Logic, NetApp, Seagate, and Symantec. We would also like to thank our colleagues in the SSRC and NetApp's Advanced Technology Group for their insightful feedback, which greatly improved the quality of the paper.

References

[1] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Feb. 2007.

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

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

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

[5] C. M. Bowman, C. Dharap, M. Baruah, B. Camargo, and S. Potti. A file system for information management. In *Proceedings of the International Conference on Intelligent Information Management Systems*, March 1994.

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

[7] *IBM DB2 Universal Database Administration Guide: Implementation*, 2000.

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

[9] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[10] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit groupings: Exploiting disk bandwidth for small files. In *Proceedings of the 1997 USENIX Annual Technical Conference*, pages 1–17, Jan. 1997.

[11] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole, Jr. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 16–25. ACM, Oct. 1991.

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

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

[14] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, Boston, MA, 1984.

[15] D. R. Hardy and M. F. Schwartz. Essence: A resource discovery system based on semantic file indexing. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 361–374, San Diego, CA, 1993.

[16] D. Hitz, J. Lau, and M. Malcom. File system design for an NFS file server appliance. In *Proceedings of the Winter 1994 USENIX Technical Conference*, pages 235–246, San Francisco, CA, Jan. 1994.

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

[18] O. Laadan, R. A. Barratto, D. B. Phung, S. Potter, and J. Nieh. DejaView: A personal virtual computer recorder. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 279–292, 2007.

[19] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the 2008 USENIX Annual Technical Conference*, Boston, MA, 2008. USENIX.

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

[21] Microsoft, Corp. WinFS: What's in store. `http://blogs.msdn.com/winfs/`, 2006.

[22] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference*, Boston, MA, 2006.

[23] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The Grid File: An adaptable, symmetric multikey file structure. *ACM Trans. on Database Systems*, 9(1):38–71, 1984.

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

[25] R. Ramakrishnan and J. Gehrke. *Database management systems*. McGraw-Hill, 3rd edition, 2003.

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

[27] S. Sechrest and M. McClennen. Blending hierarchical and attribute-based file naming. In *Proceedings of the 12th International Conference on Distributed Computing Systems (ICDCS '92)*, pages 572–580, Yokohama, Japan, 1992.

[28] S. Shah, C. A. N. Soules, G. R. Ganger, and B. D. Noble. Using provenance to aid in personal file search. In *Proceedings of the 2007 USENIX Annual Technical Conference*, Santa Clara, CA, 2007.

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

[30] C. A. N. Soules and G. R. Ganger. Connections: Using context to enhance file search. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 119–132, Brighton, UK, 2005.

[31] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: A column oriented DBMS. In *Proceedings of the 31th Conference on Very Large Databases (VLDB)*, pages 553–564, Trondheim, Norway, 2005.

[32] M. Stonebraker and U. Cetintemel. "One Size Fits All": An idea whose time has come and gone. In *Proceedings*

of the 21st International Conference on Data Engineering (ICDE '05), pages 2–11, Tokyo, Japan, 2005.

[33] 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 *Proceedings of the 33th Conference on Very Large Databases (VLDB)*, pages 23–28, Vienna, Austria, 2007.

[34] N. Yezhkova, D. Reinsel, R. Villars, R. Gray, and B. Nisbet. Worldwide disk storage systems 2007-2011 forecast: Mature, but still growing and changing. Technical Report doc-206662, IDC, 2007.