Scalable Full-Text Search for Petascale File Systems

Andrew W. Leung
Storage Systems Research Center
University of California, Santa Cruz
Santa Cruz, California 95064
Email: aleung@cs.ucsc.edu

Ethan L. Miller
Storage Systems Research Center
University of California, Santa Cruz
Santa Cruz, California 95064
Email: elm@cs.ucsc.edu

***Abstract*—As file system capacities reach the petascale, it is becoming increasingly difficult for users to organize, find, and manage their data. File system search has the potential to greatly improve how users manage and access files. Unfortunately, existing file system search is designed for smaller scale systems, making it difficult for existing solutions to scale to petascale files systems. In this paper, we motivate the importance of file system search in petascale file systems and present a new full-text file system search design for petascale file systems. Unlike existing solutions, our design exploits file system properties. Using a novel index partitioning mechanism that utilizes file system namespace locality, we are able to improve search scalability and performance and we discuss how such a design can potentially improve search security and ranking. We describe how our design can be implemented within the Ceph petascale file system.**

I. Introduction

As more business, science, government, and other entities move towards a digital infrastructure, the demand for large-scale, high-performance storage has drastically increased. This demand has resulted in an increasing number of file systems that store petabytes of data, billions of files, and serve thousands of users. Unfortunately, the scale of these file systems makes efficiently organizing, finding, and managing files extremely difficult. Users are forced to manually organize and navigate huge hierarchies, with possibly billions of files, which is very slow and inaccurate. A scientist, for example, whose simulation produces thousands of files may have to meticulously name and manually navigate thousands of files to find those with interesting data results. In petascale file systems, this manual organization and navigation often results in lost time and productivity as users try and locate files, as well as, misplaced or permanently lost data.

As file systems have grown in size, file system search has become increasingly popular because it addresses many file management problems. File system search has been an active research topic for two decades [17, 20, 38] and is becoming ubiquitous on desktop [4, 18, 33] and enterprise [15, 19, 25] file systems. Full-text (or keyword) search is the foundation of most modern file system search and the *inverted index* [22] is the primary indexing structure. An inverted index consists of a *dictionary* of keywords in the file system. Each keyword in the dictionary points to a *posting list* that contains the exact location within files where the keyword occurs, which are called *postings*. File system search alleviates many file organization, location, and management problems by allowing files to be organized and retrieved using any of their features or keywords, rather than just their pathname. As a result, much less time is spent organizing and navigating files and the risk of losing data is significantly reduced. Thus, search provides a file retrieval method that can scale with petascale file systems.

Unfortunately, providing efficient file system search at the petabyte-scale presents a number of challenges. First, the scale of these systems (billions of keywords and billions of files), makes providing fast search performance extremely difficult. Second, ensuring that search results are consistent with a large and rapidly changing file system is often slow and taxing on the file system. Third, achieving fast search and update performance without requiring lots of expensive hardware is difficult. In existing desktop and enterprise file system search tools, which only index up to millions of files [24], the dictionary can often be kept in-memory and posting lists are usually small enough to be sequential on-disk, making search and update efficient. In petascale file systems, the dictionary is too large to simply reside in-memory and must often be distributed across many machines. Likewise, long posting lists are difficult to keep sequential on-disk and can require many disk seeks to retrieve. Current file system search solutions have had very limited impact in petascale file systems because they cannot efficiently scale performance and cost with the size of these systems.

In this paper we propose the design of a novel index for petascale file system search. Unlike inverted indexes currently used for file system search that are designed for general-purpose text retrieval, we exploit the properties of petascale file systems to improve scalability and performance without requiring the use of extra hardware and which can be embedded directly within the file system. Our approach uses an index partitioning method, called *hierarchical partitioning* [30], to decompose the index into many smaller, disjoint partitions based on the file system's namespace. Through the use of an *indirect index* that manages these partitions, our approach provides flexible, fine-grained index control that can significantly enhances scalability and improve both search and update performance. In addition, we discuss how to leverage partitioning to enforce file security permissions with only a limited overhead, provide personalized search result rankings, and to distribute the index across a cluster.

Our contributions include motivating the importance of efficient search in petascale file systems, discussing an initial petascale file system search design, and describing how it can be integrated within a real-world petascale file system. Our current and future work includes the following. First, we are collecting and analyzing a data set of file keywords from several large-scale file systems. To date, no data set exists for file system search as most are targeted towards databases and the web and do not reflect file system properties. Second, we are in the process of completing our index and algorithms designs. Third, we are exploring how such a system can be implemented within the Ceph petascale file system [45] in order to evaluate its performance.

II. Background

In this section we motivate search for petascale file systems, discuss the challenges with petascale file system search, and describe related work.

A. Extended Motivation

Today's file systems can store petabytes of data, spread amongst billions of files, are composed of thousands of devices and can serve data to thousands of users. These file systems may store exabytes of data in the coming future as the digital universe is expected to expand to several zetabytes by 2011 [16].

One of the key challenges for file systems at the petascale is effectively organizing, finding, and managing the growing sea of files. Currently, file systems users are forced to manually organize and navigate huge directory hierarchies that can contain billions of files. Managing these hierarchies requires significant time and diligence by users to organize and name large numbers of files in a meaningful manner. Then users must spend more time later navigating these hierarchies with only the *hope* of finding their data. This at best this wastes time and at worst can effectively lead to data loss. In essence, *file systems lack an file retrieval method that can scale with the file system's size*. Fortunately, two decades of file system and information retrieval research have shown that file system search provides a scalable retrieval method that can mitigate many of these problems by allowing files to be retrieved using their features or keywords rather than just their pathnames [17, 20, 31, 35–38]. Similarly, full-text search has revolutionized the way web pages are organized and accessed on the Internet, demonstrating its ability to scale to very large systems.

To further motivate the importance of search in petascale file systems, we describe several use case examples taken from discussions with real large-scale file system users.

1) *Managing scientific data.* A single scientific simulation can often generate thousands of files containing experiment data. However, finding files with interesting results amongst the vast collection can be extremely difficult. As a result, scientists often take great care to name files with experiment results, such as, naming a file `run_10_succ_1h30m_22uj.data` for an experiment that was the 10^{th} run, finished successfully, took 1 hour and 30 minutes and calculated 22 microjoules of energy. Later locating results requires sifting through thousands of files to locate those with names containing useful keywords. This approach requires significant time to name and then sort through files, while not guaranteeing files will be found. However, efficient file system search can greatly ease this process, as scientists can simply issue queries for files containing the results they are interested in, also making it easier to share results with others.

2) *Archival data.* Often petascale file systems employ tiered storage architectures because storing petabytes of data on high-end disks is too expensive [32]. Files not recently used are often archived on cheaper, lower-tier tape storage. This makes retrieving archived data very difficult as data is often accessed much later, file organizations are often long forgotten, and a third party is retrieving the data [6]. As a result, finding archive data requires manual navigating the file system, which often amounts to a full scan of the file system and can take days or weeks. However, file system search allows significantly easier and faster exploration of the archive by trading slow, brute force navigation for simple and fast queries.

3) *Legal compliance data.* With increasing legal data regulations [41, 42], petascale file systems often store data that *must* be kept for a given period of time, unmodified, and be able to produce it in response to litigation. Finding and producing files when legally required to do so is extremely difficult in petascale file systems since manually navigating huge hierarchies is very slow and inaccurate. However, search enables much simpler response to litigation as files pertaining to the case can simply be recalled through a query with much higher accuracy.

These use cases represent some specific scenarios where search is extremely useful, though may not represent the most common user operations. However, several other works have shown that in most cases search is a far more intuitive and scalable method of file retrieval than traditional hierarchy navigation [14, 39, 40].

B. Petascale Search Challenges

While search is important in petascale file systems, the scale of these file systems present a number of challenges that designing an efficient solution difficult. Here were discuss some of these challenges.

1) *Cost.* Today's large-scale search engines, such as Google and Yahoo!, use large, dedicated clusters of machines to achieve high-performance [5]. Index updates are applied off-line on a separate cluster and the index is re-built weekly. However, dedicated hardware can cost millions of dollars, which is often as much as an entire file system budget, and weekly updates make file system search results too stale. Even enterprise file system search appliances can cost tens of thousands of dollars and index only millions of files [24]. Petascale file system search should require only minimal resources, reducing costs and allowing it to be integrated directly with the file system.

2) *Performance.* Most large-scale search engines and database systems make significant trade-offs between search and update performance [1, 28]. File system search must, however, strike a balance between the two, as it must quickly search through

billions of files, as well as, frequently update the index to reflect constant changes in the file system [29, 44]. As the file system scales to petabytes, the dictionary becomes too large fit in-memory and posting lists become too long to be kept sequential on-disk, resulting in numerous disk seeks for a single search. Posting list update algorithms either try and maintain on-disk sequentially at a significant cost to update performance or vice versa [28]. As a result, it is difficult to efficiently scale search and update performance without requiring significant hardware additions.
3) *Ranking.* Searching the web has been greatly improved through successful search result ranking algorithms [9]. These algorithms often only need to return the few top-K results to satisfy most queries. However, such algorithms do not yet exist for file systems, particularly, petascale file systems. Current desktop and enterprise file systems often rely on simplistic ranking algorithms that require users to sift through, possibly, thousands of search results. In petascale file systems, a single search can return millions of files, making accurate ranking critical. Previous work has looked at how to use personalization [3, 26] and semantic links [21, 37, 38] in the file system to improve accuracy.
4) *Security.* Petascale file systems often store highly secure data, such as nuclear test data. It is critical that file system search not leak privileged data. Unfortunately, current file system search tools either do not enforce file permissions or significantly degrade performance [12] to do so. In most cases, a separate index is built for each user, which can require prohibitively high disk space, or permission checks (*i.e.,*`stat()` calls) are required for every search result, which is very slow.
5) *Distributed design.* Petascale file systems are naturally distributed and can be composed of thousands of devices. This requires the index, which can grow to be 20% of the file system's size [8], to be distributed as well. Distributed file system search must be able to handle the frequent addition and removal of devices, effectively utilize file system resources while balancing load with the file system's workload.

C. Related Work

As file systems have grown in scale, more research has looked at improving file system search performance. However, only a few have looked at re-designing index structures for file systems. The Inversion file system [36] used a PostgreSQL database to index files, rather than traditional file system inode structures. Inversion allows database-style queries over files, however, general-purpose databases have only limited scalability for large-scale file system search [30]. The GLIMPSE [31] file search system uses an inverted index to route queries to parts of the namespace where results are located and `agrep` is then used to find files that match the search. While this approach greatly reduces index size, only requiring 2% to 4% of the total text size, it is much slower since many files must be read and processed. Likewise, Diamond [23] does not use an index at all, instead, using a method, called Early Discard, to more quickly scan files. As disk capacity has become much cheaper, for many systems it is preferable to sacrifice capacity and construct an index for faster search performance. The Wumpus desktop search system [10] introduces a number of improvements to conventional inverted index design, which improves full-text search on desktop file systems. However, its current design targets desktop file systems and lacks a number of features critical to petascale file system search.

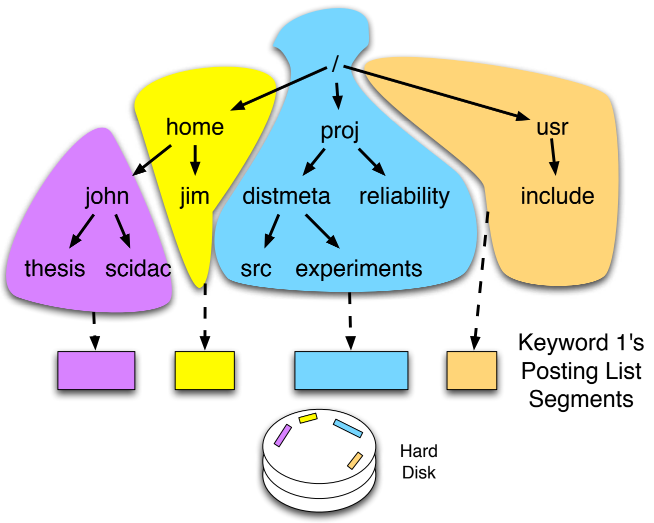

Fig. 1: ***Segment Partitioning.*** *The namespace is broken into partitions that represent disjoint sub-trees. Each partition maintains posting list segments for keywords that occur within its sub-trees. Since each partition is relatively small, these segments can be kept sequential on-disk.*

III. Our Approach

While petascale file systems present a number of challenges that make search difficult, they also have properties that can be leveraged. In particular, our inverted index design utilizes *hierarchical partitioning*, an index partition mechanism that exploits file system namespace locality [30]. Namespace locality implies that location within the file system's namespace influences the properties of files within it. That is, different sub-trees in the namespace have different access patterns (*e.g.,*frequently vs rarely accessed, metadata vs I/O workloads and read vs write workloads) [29, 43], grow at different rates [2, 13], and are often accessed by only a small fraction of users [29]. Namespace locality follows logically from the fact that the file system's namespace is already a neatly organized and classified hierarchy of files and directories, where different sub-trees usually have different uses.

A. Index Design

Our index design consists of two-levels. At the first level is a single inverted index, called the *indirect index*, that points to the locations of posting list *segments* rather than the entire posting list itself. The indirect index is similar to the inverted index used in GLIMPSE [31]. At the second level is a large collection of posting list segments. A posting list segment is

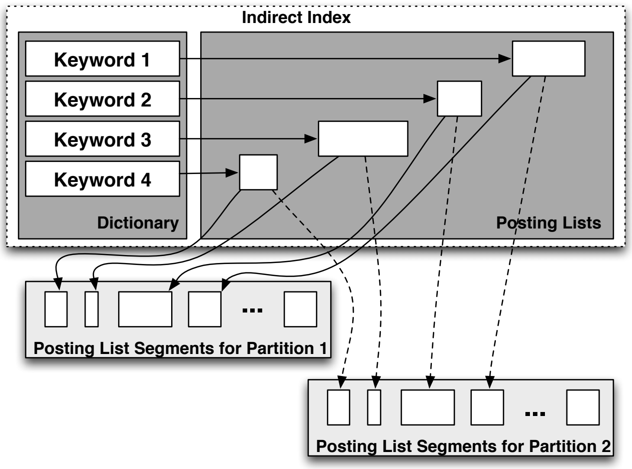

Fig. 2: ***Indirect Index Design.*** *The indirect index stores the dictionary for the entire file system and each keyword's posting lists contain locations of partition segments. Each partition segment is kept sequential on-disk.*

a region of a posting list that is stored sequentially on-disk. Posting lists are partitioned into segments using hierarchical partitioning. Thus, a segment represents the postings for a keyword that occurs within a specific sub-tree in the namespace. An illustration of how the a posting list is partitioning into segments is shown in Figure 1. The namespace is partitioned so that each sub-tree's partition is relatively small, on the order of 100,000 files. By partitioning the posting lists into segments we ensure fast performance for searching or updating any one partition, as posting lists are small enough to efficiently read, write, and cache in-memory. In essence, partitioning makes the index namespace locality-aware and allow the index to be controlled at the granularity of sub-trees.

The purpose of the indirect index is to identify which sub-tree partitions contain any postings for a given keyword. Doing so allows search, update, security, and ranking to operate at the granularity of sub-trees. The indirect index maintains the dictionary for the entire file system. The reason to maintain a single dictionary is that keeping a dictionary per-partition would simply require too much space overhead since many keywords will be replicated in many dictionaries. Each keyword's dictionary entry points to a posting list that contains the on-disk address of segments that contain actual postings. We illustrate the design of the indirect index in Figure 2. Since the indirect index only maintains a dictionary and posting lists containing segment pointers, it can be kept in-memory if properly distributed across the nodes in the file system, which we will discuss later in this section.

B. Query Execution

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

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

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

Once in-memory, segments are managed by an LRU cache. Though there have been no studies of file system query patterns, web searches [7, 27] and file access patterns [29] both exhibit Zipf-like distributions. This implies skewed popularity distributions for partitions and that an LRU algorithm will be able to keep popular partitions in-memory, greatly improving performance for common-case searches. Additionally, this enables better cache utilization since only index data related to popular partitions is kept in-memory, rather than data from all over the file system. Efficient cache utilization is important for direct integration with the file system since it will often be shared by the file system.

C. Index Updates

One of the key challenges with file system search is balancing search and update performance. Inverted indexes traditionally use either an in-place or merge-based update strategy [28]. An in-place update strategy is an update-optimized approach. When postings lists are written to disk, a sequential region on-disk is allocated that is larger than the required amount. When new postings are added to the list they are written to the empty region. However, when the region fills and new posting needs to be written, a new sequential region is allocated elsewhere on-disk and the new postings are written to it. Thus, in-place updates are fast to write since they can usually be written sequentially and do not require much pre-processing. However, as posting lists grow they become very fragmented which degrades search performance. Alternatively, a merge-based update strategy is a search-optimized approach. When a posting list is modified it is read from disk, modified in-memory, and written out sequentially to a new location. This strategy ensures that posting lists are sequential on-disk, though requires the entire posting to be read and written in order to update it, which can be extremely slow for large posting lists.

Our approach achieves a better balance in two ways. First, since posting list segments only contain postings from partitions, they are small enough to make merge-based updates efficient. When modifying a posting list, we are able to quickly read the entire list, modify it in memory, and quickly write it out sequentially to disk. Doing so keeps segment updates relatively fast and ensures that segments are sequential on-disk. An in-place approach is also feasible since small segments often will not need allocate more than one disk region. However, the space overhead from over-allocating disk regions can become quite high. Second, our approach can exploit locality in file access patterns to reduce overall disk I/Os. Often only a subset of file system sub-trees are frequently modified [2, 29]. As a result, we often only need to read segments from a small number of partitions. By reading fewer segments, far less data needs to read for an update compared to retrieve an entire posting list, cache space is better utilized, and we are able to better coalesce updates in-memory before writing them back to disk.

D. Additional Functionality

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

Secure file search is difficult because either an index is kept for each user, which requires a huge amount of disk space, or permissions for all search results need to be checked, which can be very slow [11]. However, most users only have access privileges to a limited number of sub-trees in the namespace (we plan to quantify this in future work). Hierarchical partitioning, through the use of additional metadata stored in the indirect index, can eliminate sub-trees a user cannot access from the search space. Doing so prevents users from searching files they cannot access without requiring any additional indexes and reduces the total number of search results returned, limiting the number of files whose permissions must be checked.

Ranking file system search results is difficult because most files are unstructured documents with little semantic information. However, sub-trees in the namespace often have distinct, unique purposes, such as a users home directory or a source code tree. Using hierarchical partitioning, we can leverage this information to improve how search results are ranked in two ways. First, files in the same partition may have a semantic relationship (*i.e.*,files for the same project) that can be used when calculating rankings. Second, different ranking requirements may be set for different partitions. Rather than use a one-size-fits-all ranking function for all billion files in the file system, we can potentially use different ranking functions for different parts of the namespace. For example, files from a source code tree can be ranked differently than files in a scientist's test data, potentially improving search relevance for users.

Both file system access patterns and web searches have Zipf-like distributions. Assuming these distributions hold true for file system search, a large set of index partitions will be cold (*i.e.*,not frequently searched). Our approach can allow us to identify these cold partitions and either heavily compress them or migrate them to lower-tier storage (low-end disk or tape) to improve cost and space utilization. A similar concept has been applied to legal compliance data in file systems and has shown the potential for significant space savings [34].

E. Integration within Ceph

Since addressing file system search performance with additional hardware can be prohibitively expensive at the petascale, it is important search can be efficient integrated within the file system. File system search should provide scalable performance while not interfering with normal file operation. We discuss how our approach can be integrated with the Ceph petascale file system [45].

Ceph is an object-based parallel file system. A cluster of metadata servers (MDSs) handles metadata operations while a cluster of object storage devices (OSDs) handles data operations. The MDS cluster manages the namespace and stores all file metadata persistently on the OSD cluster. Since the OSD cluster is used for metadata storage, MDS nodes can generally afford a significant amount of main memory (tens of gigabytes).

We intend for the indirect index to be distributed across the MDS cluster and across enough nodes so that it can be kept in-memory. Since a significant amount of query pre-processing takes place in the indirect index, keeping it in-memory will significantly improve performance. Posting list segments will be stored on the OSD cluster and since they are small they can map directly to physical objects. The indirect index will be partitioned across the MDS cluster using a *global inverted file* (IF_G) partitioning approach. In this approach keywords are used to partition the index such that each MDS stores only a subset of the keywords in the file system. For example, with two MDS nodes A and B, A may index and store all keywords in the range $[a - q]$ and B may index and store all remaining keywords. Using an IF_G partitioning approach limits network bandwidth requirements as messages are sent only to the MDS nodes responsible for query keywords.

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

IV. Conclusions and Future Work

As file systems continue to grow, scalable file retrieval is emerging as one of the key challenges. As a result, file system search has becoming increasingly popular because it greatly improves how users organize and retrieve files. Unfortunately, existing file system search solutions have difficulty scaling cost and performance to petascale file systems. To address this problem, we presented an initial petascale file system search design. Our design exploits file system properties to improve search scalability and performance while not requiring significant hardware additions. While we presented some initial ideas, there is much future work.

1) To the best of our knowledge no large-scale file system keyword data sets exist. Those available, such as those from TREC, are designed for database and web search. We are planning to collect keyword data sets from real-world large-scale file systems using a secure approach that anonymizes keywords while preserving namespace locality.
2) Our design assumes that file system keywords exhibit namespace locality, which has been shown to be the case for file metadata [2, 30]. To validate and quantify keyword namespace locality we plan on analyzing the keyword data sets we collect.
3) We have presented a number of initial index designs. However, our design is far from complete. We will use the results of our keyword analysis to guide and finalize our design.
4) To better understand how search and file systems can be integrated we are planning to implement and evaluate our design in the Ceph petascale file system.

Acknowledgements

This work was supported in part by the Department of Energy under award DE-FC02-06ER25768 and by the industrial sponsors of the Storage Systems Research Center (SSRC), including Data Domain, DigiSense, LSI Logic, NetApp, Seagate, and Symantec. We thank Shankar Pasupathy, Tim Bisson, Minglong Shao, and the other members of NetApp's Advanced Technology Group, as well as, the members of the SSRC, whose advice helped guide this research. Finally, we thank the anonymous reviewers for their insightful feedback.

References

[1] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-Stores vs. Row-Stores: How different are they really?" in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, Vancouver, BC, Canada, June 2008.

[2] 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 '07)*, Feb. 2007, pp. 31–45.

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

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

[5] A. Arasu, J. Cho, H. Garcia-Molina, A. Paepcke, and S. Raghavan, "Searching the web," *ACM Transactions on Internet Technology*, vol. 1, no. 1, pp. 2–43, 2001.

[6] M. Baker, M. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungale, "A fresh look at the reliability of long-term digital storage," in *Proceedings of EuroSys 2006*, Apr. 2006, pp. 221–234.

[7] 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 *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in informaion retrieval (SIGIR '04)*, Sheffield, UK, June 2004.

[8] T. C. Bell, A. Moffat, C. G. Nevill-Manning, I. H. Witten, and J. Zobel, "Data compression in full-text retrieval systems," *Journal of the American Soiety for Information Sciences*, vol. 44, no. 9, pp. 508–531, 1993.

[9] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[10] S. Büttcher, "Multi-user file system search," Ph.D. dissertation, University of Waterloo, 2007.

[11] S. Büttcher and C. L. A. Clarke, "A security model for full-text file system search in multi-user environments," in *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05)*, San Francisco, CA, Dec. 2005, pp. 169–182.

[12] S. Buttcher and C. L. Clarke, "A security model for full-text file system search in multi-user environments," in *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05)*, San Francisco, CA, December 2005.

[13] J. R. Douceur and W. J. Bolosky, "A large-scale study of file system contents," in *Proceedings of the 1999 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1999.

[14] D. Ellard, "The file system interface in an anachronism," Harvard University, Tech. Rep. TR-15-03, November 2003.

[15] Fast, "FAST – enterprise search," http://www.fastsearch.com/, 2008.

[16] J. F. Gantz, C. Chute, A. Manfrediz, S. Minton, D. Reinsel, W. Schlichting, and A. Toncheva, "The Digital and Exploding Digital Universe: An updated forecast of worldwide information growth through 2011," International Data Corporation (IDC), Tech. Rep., March 2008.

[17] 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)*. ACM, Oct. 1991, pp. 16–25.

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

[19] ——, "Google enterprise," http://www.google.com/enterprise/, 2008.

[20] 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)*, Feb. 1999, pp. 265–278.

[21] K. Gyllstrom and C. Soules, "Seeing is Retrieving: Building information context from what the user sees," in *Proceedings of the 2008 International Conference on Intelligent User Interfaces*, Maspalomas, Gran Canaria, Spain, January 2008.

[22] D. Harman, R. Baeza-Yates, E. Fox, and W. Lee, *Information retrieval: data structures and algorithms*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992, ch. Inverted files, pp. 28–43.

[23] 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 *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST '04)*. San Francisco, CA: USENIX, Apr. 2004, pp. 73–86.

[24] G. G. Inc., "Compare search appliance tools," http://www.goebelgroup.com/sam.htm, 2008.

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

[26] J. Koren, Y. Zhang, S. Ames, A. Leung, C. Maltzahn, and E. L. Miller, "Searching and navigating petabyte scale file systems based on facets," in *Proceedings of the 2007 ACM Petascale Data Storage Workshop (PDSW 07)*, Reno, NV, November 2007.

[27] R. Lempel and S. Moran, "Predictive caching and prefetching of query results in search engines," in *Proceedings of the 13th International World Wide Web Conference*, Budapest, Hungary, 2003.

[28] N. Lester, A. Moffat, and J. Zobel, "Efficient online index construction for text databases," *ACM Transactions on Database Systems*, vol. 33, no. 3, 2008.

[29] 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*, Jun. 2008.

[30] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller, "Spyglass: Fast, scalable metadata search for large-scale storage systems," in *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST '09)*, February 2009.

[31] U. Manber and S. Wu, "GLIMPSE: A tool to search through entire file systems," in *Proceedings of the Winter 1994 USENIX Technical Conference*, San Francisco, CA, 1994.

[32] J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg, "IBM Storage Tank—a heterogeneous scalable SAN file system," *IBM Systems Journal*, vol. 42, no. 2, pp. 250–267, 2003.

[33] Microsoft, Inc., "Windows search 4.0," http://www.desktop.google.com/, 2007.

[34] S. Mitra, M. Winslett, and W. W. Hsu, "Query-based partitioning of documents and indexes for information lifecycle management," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, Jun. 2008.

[35] 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.

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

[37] 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*, Jun. 2007, pp. 171–184.

[38] 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)*. New York, NY, USA: ACM Press, 2005, pp. 119–132.

[39] C. A. Soules and G. R. Ganger, "Why can't i find my files? new methods for automatic attribute assignment," in *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*, Sydney, Australia, 1999.

[40] J. Teevan, C. Alvarado, M. S. Ackerman, and D. R. Karger, "The perfect search engine is not enough: a study of orienteering behavior in directed search," in *Proceedings of the 2004 Conference on Human Factors in Computing Systems (CHI '04)*. ACM Press, 2004, pp. 415–422.

[41] United States Congress, "The health insurance portability and accountability act (hipaa)," http://www.hhs.gov/ocr/hipaa/, 1996.

[42] ——, "The sarbanes-oxley act (sox)," http://www.soxlaw.com/, 2002.

[43] W. Vogels, "File system usage in Windows NT 4.0," in *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, Dec. 1999, pp. 93–109.

[44] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, D. D. E. Long, and T. T. McLarty, "File system workload analysis for large scale scientific computing applications," in *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, College Park, MD, Apr. 2004, pp. 139–152.

[45] 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 *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. Seattle, WA: USENIX, Nov. 2006.