Dynamic Non-Hierarchical File Systems for Exascale Storage

Applicant/Institution:
Street Address/City/State/Zip:

Principal Investigator: Darrell D. E. Long
Postal Address: Computer Science Department
Jack Baskin School of Engineering
University of California
Santa Cruz, CA, 95064
Telephone Number:
Email: darrell@cs.ucsc.edu

Funding Opportunity Announcement Number: DE-FOA-0000256
DOE/Office of Science Program Office: Office of Advanced Scientific Computing Research
DOE/Office of Science Program Office Technical Contact: Dr. Lucy Nowell

Budget Table:

Investigator	Institution	Amount Requested ($)		
		Year 1	**Year 2**	**Year 3**
Long, Darrell (PI)	UCSC	37,696	38,826	39,991
Miller, Ethan (Co-PI)	UCSC	13,146	13,539	27,891

8 PROJECT NARRATIVE

8.1 Project Objectives

The ultimate goal of this project is to improve data management in scientific computing and high-end computing (HEC) applications, and to achieve this goal we propose:

- To develop the first, HEC-targeted, file system featuring rich metadata and provenance collection, extreme scalability, and future storage hardware integration as core design goals.
- To evaluate and develop a flexible non-hierarchical file system interface suitable for providing more powerful and intuitive data management interfaces to HEC and scientific computing users.

8.2 Introduction and Motivation

Data management is swiftly becoming a serious problem in the scientific community – while copious amounts of data are good for obtaining results, finding the *right* data is often daunting and sometimes impossible. Scientists participating in a Department of Energy workshop noted that most of their time was spent "...finding, processing, organizing, and moving data and it's going to get much worse" [2]. Indeed, the Large Synoptic Survey Telescope (LSST) [99] is expected to produce around 20 terabytes of data every night, obtaining repeat exposures of the entire night sky. Scientists should not be forced to become data mining experts in order to retrieve the data they want, nor should they be expected to remember the naming convention they used several years ago for a set of experiments they now wish to revisit. Ideally, locating the data you need would be as easy as browsing the web.

Unfortunately, existing data management approaches are usually based on hierarchical naming, a 40 year-old technology designed to manage thousands of files, not exabytes of data. Today's systems do not take advantage of the rich array of metadata that current high-end computing (HEC) file systems can gather, including content-based metadata and provenance[1] information. As a result, current metadata search approaches are typically *ad hoc* and often work by providing a parallel management system to the "main" file system, as is done in Linux (the `locate` utility), personal computers [11], and enterprise search appliances [35, 56]. These search applications are often optimized for a single file system, making it difficult to move files and their metadata between file systems. Users have tried to solve this problem in several ways, including the use of separate databases to index file properties [91], the encoding of file properties into file names, and separately gathering and managing provenance data [39], but none of these approaches has worked well, either due to limited usefulness or scalability, or both.

We propose an integrated data storage system, targeted at exascale systems, with improved content management and an improved interface. By integrating metadata, file content, and provenance into a single system, any changes made can be indexed immediately, significantly increasing the power and usefulness of a subsequent search. Such searches, in turn, would allow users, and this is particularly relevant for scientific computing users [2], to spend more time accessing their data, as opposed to locating and moving it. For example, imagine that we find that program Y has a bug. A user might want to find all output files from experiment X containing data tainted by the buggy program, so that the files could be reprocessed with a corrected version of program Y. This query requires both provenance and tag-based metadata; the provenance identifies the files tainted by Y and the tag-based metadata identifies the files that are part of experiment X.

Richer metadata and provenance information, integrated into the underlying file system, allows for more flexible and user-oriented access to the data. For example, enhanced metadata and provenance can be leveraged to provide statistics about the data in the system, aiding in more accurate query results. Retrieving statistics based on the metadata and provenance of the data could let us dynamically create signature files which are completely up to date with the current state of the system. In this way, the search space can

[1] *Provenance*, or lineage, is information about the inputs and processes used to create a particular file.

accurately be narrowed to only results matching the user's query. Since the additional functionality and metadata is provided as an integral part of the file system, there is no longer a need for names to be static paths: directories and file names can be queries into an index. This provides multiple ways to access the same piece of data, freeing the user from remembering an arbitrary static path. The benefits of this approach go beyond an improved mechanism to deal with file naming challenges, it also offers a more flexible and efficient interface for different user classes. For example, if the same underlying data set is being used by multiple scientists from different fields, a dynamic naming scheme would allow each scientist to be provided with access to the data in a customized fashion optimized to their distinct purposes.

Without such enhanced functionality in a scalable system, scientific and HEC users are in desperate need of improved data management, particularly if HEC systems are to scale and remain usable. Existing technologies are both too specialized and impractical for exascale computing [2]. No existing file system is suitable, and any attempt to simply layer functionality atop existing HEC file systems comes with a performance penalty, which such systems are least prepared to tolerate. We propose a novel solution building on proven technology that we have developed to design a new file system with the required functionality an integral part of file system design and not an added afterthought to an existing infrastructure. By pursuing a file system we aim to offer the most general data management solution possible to scientific computing users, without the added overheads of ill-matched database systems [2]. Our goal in taking this approach is to provide an exascale data storage system that can be easily built upon to meet specific HEC user needs thanks to improved functionality and richer metadata, features that would simultaneously allow the support of legacy data and interfaces.

8.3 Proposed Research

Our goal in developing this system is twofold: we want to hide and improve the operation of the system through automation and improved data management, and we wish to target and extend the facilities available to scientists through improved metadata and provenance collection, management, and interfaces. The proposed solution looks at designing a novel file system architecture that integrates file content, metadata, and provenance. A new architecture is long overdue – few people today would want to entrust their life savings to a bank whose systems haven't been updated since 1970; nor should scientists have to entrust their data to a similarly outdated architecture. Existing high performance file systems offer general-purpose usefulness, but not the provenance and enhanced functionality needed to reduce the data management effort of users. While this missing functionality could be provided by relational database systems, or through the layering of the functionality atop existing file systems, neither solution offers enough flexibility to be of general use, or the necessary performance to be suitable for exascale computing. To provide HEC users with a general purpose data management system that is functional, flexible, and of general usefulness (including compatibility with legacy data and file system interfaces) we propose to develop a novel file system that includes the requisite provenance and rich metadata management from the offset.

To develop such a file system, several major goals need to be achieved. Specifically:

- High-performance, real-time metadata harvesting: extracting important attributes from files dynamically and immediately updating indexes used to improve search.
- Transparent, automatic, and secure provenance capture: recording the data inputs and processing steps used in the production of each file in the system.
- Scalable indexing: indexes that are optimized for integration with the file system.
- Dynamic file system structure: our approach provides dynamic directories similar to those in semantic file systems, but these are the *native* organization rather than a feature grafted onto a conventional system.

In addition to these goals, our research effort will include evaluating the impact of new storage technologies on the file system design and performance. In particular, the indexing and metadata harvesting functions

can potentially benefit from the performance improvements promised by new storage class memories. The efficient use of such technologies in storage systems software is a current and active research effort for the PIs [8,38,55,70], and our design will be executed in anticipation of upcoming opportunities and changes to the storage hardware landscape.

To meet the above goals, we will address four major research areas: metadata management; provenance; scalable indexing; and non-hierarchical name spaces. The maintenance of richer metadata presents challenges to the efficient representation, storage, and retrieval of such information. Provenance in particular is a particularly useful form of metadata, and offers the added challenge and opportunity of automating its capture and its efficient representation. All the additional metadata being collected and stored will need to be indexed and located efficiently, as the system grows to exabyte scales. Finally, once all this enhanced metadata has been efficiently captured, stored, and indexed, it is essential to effectively provide the most flexible and useful interface to the new file system, and to this end we will leverage and extend the state of the art in non-hierarchical namespaces.

8.4 Metadata Management

File system metadata should be treated as an aid to managing and accessing data and not a rigid and limited structure to which the user must conform. To this end we propose to enhance metadata management to provide seamless support for a search-based dynamic interface to the files. File system search provides a clean, powerful abstraction from the file system. It is often easier to specify *what* one wants using file metadata and extended attributes rather than specifying *where* to find it [88]. Searchable metadata allows users and administrators to ask complex, ad hoc questions about the properties of the files being stored, helping them to locate, manage, and analyze their data.

Unfortunately, metadata search applications face several limitations in exascale systems. First, search applications must track all metadata changes in the file system, a difficult challenge in a system with billions of files and constant metadata changes. Second, indexes must be kept up-to-date, with changes reflected immediately, to prevent a search from returning very inaccurate results. Keeping the metadata index and "real" file system consistent is difficult because collecting metadata changes is often slow [51,92] and search applications are often inefficient to update [3]. Third, search applications often require significant disk, memory, and CPU resources to manage larger file systems using the same techniques that are successful at smaller scales. Thus, a new approach is necessary to scale file system search to large-scale file systems.

Our approach to metadata management will be designed and developed in the context of a HEC file system that separates metadata service from bulk data service, such as PVFS [21], Lustre [85], or Ceph [100]. By following this approach, we can test our techniques in a prototype metadata server (MDS) that can be used as a drop-in replacement for the existing MDS. We may need to add hooks to gather information—provenance and content-based metadata, for example—into the client file system component as well, but the majority of the techniques can be implemented in a separate server that we will make available to the HEC community.

When designing our system there are two goals we are aiming to achieve. First, we want a metadata organization that could be quickly searched. Second, we want to provide the same metadata performance and reliability that users have come to expect in other high performance file systems. We focus on the problems that make current designs difficult to search, leaving other useful metadata designs intact. Our design leverages metadata specific indexing techniques we developed in Spyglass [61]. We plan to build upon the technologies we are in the process of developing:

- The use of a search-optimized metadata layout that clusters the metadata for a sub-tree in the namespace on disk to allow large amounts of metadata to be quickly accessed for a query.
- Efficient routing of queries to particular sub-trees of the file system using Bloom filters [15].

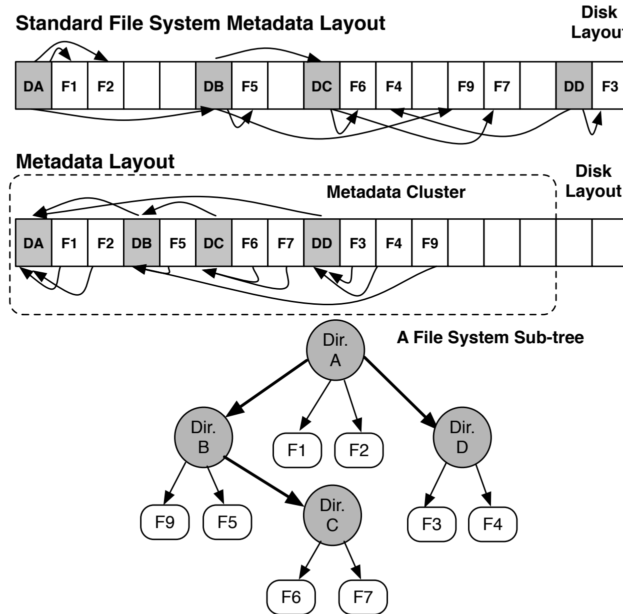


Figure 1: ***Metadata clustering.*** *Each block corresponds to an inode on disk. Shaded blocks labeled 'D' are directory inodes while non-shaded blocks labeled 'F' are file inodes. In the top disk layout, the indirection between directory and file inodes causes them to be scattered across the disk. The bottom disk layout shows how metadata clustering co-locates inodes for an entire sub-tree on disk to improve search performance.*

- The use of metadata journaling to provide good update performance and reliability for our search-optimized designs.

8.4.1 Metadata Clustering

Accessing metadata often requires numerous disk seeks to access the file and directory inodes, limiting search performance. Though file systems attempt to locate inodes near their parent directory on disk, inodes can still be scattered across the disk. For example, FFS stores inodes in the same on disk cylinder group as their parent directory [67]. However, prior work has shown that inodes for a directory are often spread across multiple disk blocks. Furthermore, directory inodes are not usually adjacent to the first file inode they name, nor are file inodes often adjacent to the next named inode in the directory [30]. We illustrate this concept in the top part of Figure 1, which shows how a sub-tree can be scattered on disk.

We propose addressing this problem using metadata clustering, a concept similar to embedded inodes [30] which store file inodes adjacent to their parent directory on disk. Metadata clustering goes a step further and stores a group of file inodes and directories adjacent on disk. Metadata clustering exploits several file system properties. First, disks are much better at sequential transfers than random accesses. Metadata clustering leverages this to pre-fetch an entire sub-tree in a single large sequential access. Second, file metadata exhibits *namespace locality*: metadata attributes are dependent on their location in the namespace [6, 61]. For example, files owned by a certain user are likely to be clustered in that user's home directory or their active project directories, not randomly scattered across the file system. Thus, queries will often need to search files and directories that are nearby in the namespace. Clustering allows this metadata to be accessed more quickly using fewer I/O requests. Third, metadata clustering works well for many file system workloads, which exhibit similar locality in their workloads [63, 83]. Often, workloads access multiple, related directories, which clustering works well for.

While clustering can improve performance by allowing fast pre-fetching of metadata for a query, it can negatively impact performance if clusters become too large, since such clusters that are too large waste disk bandwidth by pre-fetching metadata for unneeded files. We continue to research optimal cluster size and

clustering algorithms that re-balance cluster distributions over time.

8.4.2 Query Execution

Searching through many millions of files is a daunting task, even when metadata is effectively clustered on disk and indexed. Fortunately, as we mentioned in Section 8.4.1, metadata attributes exhibit namespace locality, which means that attribute values are influenced by their namespace location and files with similar attributes are often clustered in the namespace.

We will leverage clustering by keeping summaries of attributes for each cluster in a set of *signature files* [27]—compact bit arrays describing the contents of the cluster—and searching the signature files for all clusters to identify specific clusters that *might* contain results for the query because their signature files indicate that files in that cluster have *all* of the desired attribute-value pairs. Because signature files, like Bloom filters [15], use hashing to set bit positions and because signature files do not capture information about whether a combination of attribute-value pairs is present in a single file, there is the potential for false positives; however, the use of signature files allows the MDS to quickly rule out the vast majority of metadata clusters and apply the more computationally-expensive full search and similarity engines to only those clusters that potentially have relevant results for the current query, dramatically improving scalability. Even this has trouble scaling to exascale systems; we will investigate applying hierarchical clustering techniques to narrow the search space with broad Bloom filters at higher levels.

8.4.3 Cluster-based Journaling

Search-optimized systems organize data so that it can be read and queried as fast as possible, often causing update performance to suffer [3,98]. However, file systems require *both* fast search performance and rapid update. Our approach facilitates fast update by combining logging and index partitioning. Logging allows the system to delay updates to the cluster's primary, highly-compressed index at the cost of more expensive searches of recent metadata updates in the log. By maintaining a per-cluster log, however, we limit the amount of logged information that must be searched.

Logging updates allows users to pose "time-traveling" queries over past metadata versions, helping them gain information about how and when their files changed. As metadata is added to a cluster's log, the signature files for the cluster are updated; thus, index searches need only consider logs for clusters that might contain relevant results. Logging also allows updates to the index to be batched, amortizing the cost of rebuilding the optimized index for the cluster across many metadata modifications. We expect that file activity will be concentrated in relatively few clusters at any given time; thus, our approach of partitioning the log as well as the primary index will aid in scalability. While updates to existing files are sent to the cluster that already contains metadata for the file, techniques for deciding *which* cluster receives new files' metadata are discussed in Section 8.6.3.

8.5 Provenance: Effective Attribute and Provenance Collection

Provenance information is a critical part of file system metadata information, since it provides insight into the processes that created a particular piece of data. Previous work has demonstrated that provenance information can be efficiently captured and stored for relatively small data collections [75,76]; we will expand on this work to integrate provenance collection and management across exabyte-scale storage with billions of files. We will also address the security issues that were identified in earlier work, specifically, securing provenance [16,76], using provenance to prove authenticity [43], and dealing with implicit information flows. In some cases, provenance requires more restrictive access controls than the data it describes and sometimes it requires less restrictive controls. In the absence of a consistent default assignment of provenance access controls, we must address the challenge of designing a suitable security system and making it intuitive and easy to use. Regardless of the restrictiveness of the access controls, we also need to ensure that provenance is tamper-proof, so that adversaries cannot revise history by changing provenance. Finally, while most provenance systems capture explicit information flow, implicit flows (things that result from

events *not* happening) pose a particularly significant challenge. Current systems record object reads only if they later lead to writes, but for completeness, we would need to capture all reads, regardless of whether they led to later writes or not. Unfortunately, the overhead of doing so is prohibitive, thus an open question is determining an acceptable compromise. We will develop techniques for managing provenance at exabyte-scale along with other metadata, for guaranteeing integrity of provenance that records sufficient operations to recreate both the inputs and the workflow that generated a particular piece of data, and for efficiently extracting file attribute and inter-file relationship information from the file system.

There are several key challenges that need to be addressed to meet this goal. First, there must be an infrastructure in place for extracting information from the file system. This infrastructure must be able to gather both information from files themselves and the context in which they are used. Second, this infrastructure must be fast and efficient. It is critical that extracted information be closely consistent with the file system because the query interface is the *only* naming interface in the file system. Thus, our system must be able keep up with the rate of modifications to the file system. Third, extracting information must not significantly degrade native file system performance; crawling file content or monitoring inter-file operations may utilize file system resources, potentially diminishing file system throughput.

We are developing new scalable techniques for extracting file metadata. Basic metadata—information currently stored in inodes and small-scale extended attributes—are easily handled by our approach, since updates can be sent directly to the MDS. Since they are relatively small, it is likely that they will only be updated by a single client even for a large file; thus, handling such updates requires relatively few MDS resources. However, file content in a HEC system is orders of magnitude larger than basic metadata [6], and extracting metadata from HEC files can be computationally and I/O intensive. Thus, we will rely upon *transducers* that are customized to each type of file. Content indexing can be done in one of two ways: MDS-driven or client-driven. For MDS-driven indexing, the MDS can easily note files that have been opened for writing and issue requests to have transducers run on them. By targeting content extraction to just those files, our system can quickly derive new metadata for modified files. Alternatively, the client could start the transducer as needed; we will explore tradeoffs between these two approaches.

It is impractical for our team to write transducers for every type of file that might be encountered; instead, we will adopt an approach similar to that used by Spotlight [12]: our system will provide an API that transducers can use to send content-derived metadata to the MDS. Unlike Spotlight, however, our system must accommodate per-file metadata that comes from multiple nodes simultaneously, since it is likely that a transducer will run on multiple client nodes or multiple data server nodes. We will explore approaches that coalesce metadata *before* it reaches the MDS, reducing the network, I/O and CPU load on the MDS.

Another critical issue is the ability to extract file relationships and context information from the file system, including information such as provenance [76, 89] and temporal context [93]. In our system, relationships between files are *first-class* entities, and can have tags and ownership information associated with them. Tracking such relationships requires monitoring calls to the file system (*e.g.*, `open()` and `close()`) and providing additional API calls to explicitly establish relationships between files [8]. In addition to providing API calls to explicitly link files, our system will implicitly gather provenance and other relationship information by monitoring processes on client nodes, and will coalesce that information either at the client level or on the MDS. Coalescing before the information reaches the MDS has the advantage of reducing the MDS load, but may be more difficult because of the need for many-to-many communication to do so.

We will explore the use of upcoming storage-class memory technologies and novel integrations of these technologies with log-structuring of metadata. In order to improve the latency, and potentially throughput, of metadata gathering, we will store incoming metadata in a non-volatile log. This approach is common in databases [37] and file systems [46]; we will explore the use of non-volatile memories such as flash and phase-change RAM (also called *storage class memories*) to hold the log, greatly improving performance over disk-based approaches. Using a log has a second advantage: it allows the system to retain older

versions of metadata, allowing users to do searches over historical information, to answer questions about which files have changed, and how they have changed.

8.6 Scalable Indexing

One major challenge we will address is the construction of a scalable indexing structure capable of handling the billions of files stored by an exascale computer. This metadata index must be distributed, both because of the amount of data it must manage and because of the speed at which it must run. The index must handle both updates and queries quickly—even at high throughput, a latency longer than 10–20 ms will be noticed by users. While existing techniques can be used to build a high-performance index for hierarchical data [61], significant challenges remain in building an index that can handle file metadata, content-based metadata, tags, and provenance. Our research will explore tradeoffs in partitioning, distribution, and replication that allow the metadata index to scale to thousands of requests per second across billions of files. Unlike Web indexes, our metadata indexes must be dynamic, both because of newly created provenance data and because of varying metadata and views that different users may have; both of these dynamisms can be used to improve the quality of search results. In addition, we will explore approaches that allow the system to efficiently harvest file metadata and provenance for inclusion in the index.

Providing good performance and scalability for file search in an exascale system requires new indexing solutions. Existing file search systems rely on general-purpose, off-of-the-shelf solutions, such as relational database systems [91]. However, these systems are typically designed for other workloads, such as business processing, and thus make few file search optimizations [94, 95]. As a result, they lack the performance and scalability to address file search at large scale. In addition, the large scale of HEC systems makes it impractical to address performance by simply adding more hardware, both because supplying hardware at the rate of data growth is often prohibitively expensive, and database search often does not scale linearly in the number of nodes. Instead, a file system search index must leverage file system and workload properties to achieve the necessary performance.

Existing solutions handle relationships between files poorly, if at all. However, our approach to metadata requires that we efficiently store and index relationships ranging from explicitly-defined and provenance-dictated links, to inferred relationship links. Thus, our index must efficiently store a directed graph with billions of nodes across a distributed set of metadata servers and allow that graph to be searched and updated in milliseconds. Meeting these goals is a key challenge for this research.

8.6.1 Indexing Metadata and Content

The approach we are proposing provides an index for two types of per-file metadata: simple metadata and content-based metadata. Simple metadata is structured as $\langle attribute, value \rangle$ pairs, with attributes including inode fields (*e.g.*, size, owner, timestamps) and extended attributes (*e.g.*, document title, experiment parameters, funding information). Short tags derived from content are considered simple metadata, since they are small and readily indexed using the same techniques as other simple metadata. Content-based metadata, however, is quantitatively different from simple metadata: it can represent megabytes of data per file. For example, the content-based metadata for a text file might include a postings list [101]—a set of $\langle term, count \rangle$ pairs for each term (word) in the document, and the content-based metadata for a climate simulation might include a low-resolution map or set of climate conditions at specific points from various time samples.

Searching against the different types of metadata also uses a different process. Simple metadata can be compared using relatively static evaluations, similar to those provided in a relational database system: string equality, regular expressions, and numeric comparisons. However, content-based metadata is often evaluated for *similarity* to an exemplar; for example, a user might search for climate simulation results similar to the one she just produced. Similarity evaluation requires more computation, and may also require application-specific routines to compute a similarity metric. Thus, we will provide an API, akin to that for transducers to provide metadata, that allows HEC users to supply routines to evaluate similarity between

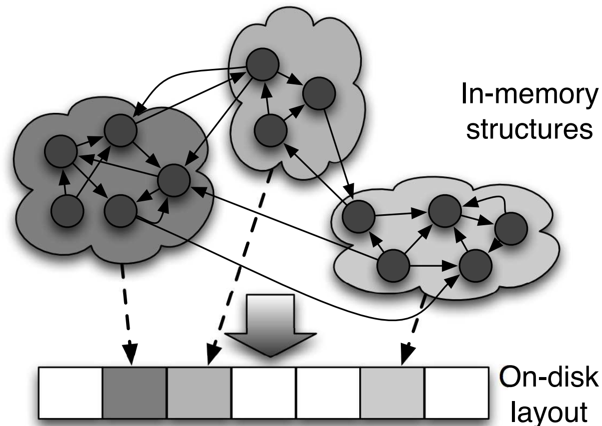


Figure 2: ***Metadata clustering example.*** *Metadata clusters, partitions, shown in different shades, index different partitions of file system metadata. Each partition is stored sequentially on disk. The entire file system metadata index is composed of the set of all metadata clusters.*

multiple files of similar type. As with transducers, it will be impractical for our team to write all possible similarity engines; in order to test our approach, we will develop several similarity engines for plain text files and one or two other sample HEC applications.

To provide reasonable search performance, we propose that metadata be partitioned into clusters, as discussed in Section 8.4.1. Our system will maintain multiple search structures for each cluster, leveraging existing research in distributed metadata [25, 79, 100] and expanding it to efficiently handle billions of files. In our approach, one structure in each cluster indexes all of the "simple" metadata; we are currently using a *kd*-tree [14] to store this information because it provides fast multidimensional search across multiple attributes. K-D trees are similar to binary trees, though different dimensions are used to pivot at different levels in the tree. K-D trees allow a single data structure to index all of a cluster's metadata. A one-dimensional data structure, such as a B-tree, would require an index for each attribute, making reading, querying, and updating metadata more difficult.

While K-D trees are good for multi-attribute queries, they are less efficient for some common operations. Many file systems, such as Apple's HFS+ [10], index inodes using just the inode number, often in a B-tree. Operations such as path resolution that perform lookups using just an inode number are done more efficiently in a B-tree than a K-D tree that indexes multiple attributes, since searching just one dimension in a K-D tree uses a range query that requires $O(kN^{1-1/k})$ time, where k is the number of dimensions and N is the size of the tree as compared to a B-tree's $O(\log N)$ look up. We will therefore continue to explore different data structures for efficient search.

Content-based metadata in the cluster, in contrast, must be stored in optimized structures particular to the needs for each similarity engine. Techniques for storing inverted indexes for text are well-known [59, 101], and typically involve a dictionary of in the partition along with a postings list that contains the exact location (file, and perhaps offset within the file) of each occurrence of a term. Satisfying a query thus involves a mathematical function across sets of postings lists for different terms; many such functions have been proposed. Our challenge is to provide the flexibility to allow different similarity engines to manage their *own* optimized structures, since non-text similarity may require different approaches to storing "terms." Meeting the challenges in providing a general indexing mechanism for non-text content-based indexing by allowing similarity engines to manage both an optimized primary search structure and a log with updates is one of our major research goals.

Individual cluster indexes will be stored sequentially on disk or other non-volatile medium, allowing them to be quickly read into RAM and decompressed on the fly. Our earlier work has shown that file

system activity, including queries, tends to be focused in relatively small parts of the file system at any given time, though the specific areas of interest vary over time. Thus, caching is likely to be highly effective at ensuring that relevant partitions are already loaded into RAM and ready for high-speed search. However, the effectiveness of different metadata attributes at limiting the scope of full search to a few clusters and improving the effectiveness of caching is an open question that we will explore.

We have some background in evaluating the effectiveness of attributes on limiting the scope of clusters to be searched, specifically the file ownership and permissions attribute. Security Aware Partitioning is a new algorithm we developed to support fast, scalable search, while maintaining the security of files. Since not every user has access to every file, displaying search results that are not accessible to a user is both insecure and poor user interface design. A list of results should only contain documents which are accessible to the user initiating the search. Unlike other partitioning schemes, which must apply an expensive filtering operation after results have been collected to enforce these restrictions, Security Aware Partitioning takes these requirements into account when building the partitions. This increases search efficiency and can prevent statistical attacks on ranked search, such as the one demonstrated by Büttcher [19]. In our partitioning scheme, if someone has permission to access one file in a partition, he can access every file in that partition. However, it is necessary to determine access in a way appropriate for the security model of the underlying file system. More details and preliminary results are discussed in Section 8.9. While these results are hopeful, we will continue to evaluate the effectiveness of other attributes as well, including explicitly-supplied tags, file type, and others on limiting the scope of clusters that must be searched.

We will also expand on the functionality provided by file system indexes, providing features not typically available in current file systems and search indexes. There are several critical components to make search practical for everyday, real-world use. First, search must enforce file security, however, doing so efficiently is not straightforward [13, 19]. Our techniques allow security information to be used during index partitioning and embedded within each partition. Doing so allows us to eliminate partitions with improper permissions from the search space, improving performance and potentially altering the ordering of returned results. Second, search must allow the combination of per-file metadata with graph-based information, described in Section 8.6.3, to permit searches that find relevant "nearby" files. This functionality is not present in current Web-based searches: there is no way to find a page to which a given page links that contains a certain term, for example.

8.6.2 Indexing File Content

In addition to searching file metadata, most users need to be able to quickly search the contents of their files. Unlike metadata, which is structured $\langle$*attribute*, *value*$\rangle$ pairs, file content is often a collection of unstructured keywords, and there are often orders of magnitude more keywords associated with a file than there are attributes.

Content search performance is limited by dictionary lookup and posting list retrieval times. Unfortunately, achieving good performance in a large file system is difficult for two key reasons. First, the scale of these systems makes retrieval a challenge because the dictionary can contain billions of keywords and posting lists can be long and fragmented on disk. Second, extracting keywords from many rapidly changing files and modifying on disk posting lists is often slow and can tax the file system.

We are addressing these difficulties with new inverted index designs that are tailored for large-scale file systems. This first entails an analysis of keyword distributions in large file systems. A keyword analysis provides us with valuable insights into what an effective design should contain. Using this information we will develop new inverted index designs with storage layouts and retrieval mechanisms that are more efficient for file system search workloads [62]. Techniques similar to those used for metadata search [61], such as hierarchical partitioning, may be helpful. Partitioning can break long term posting lists into smaller, sequentially stored units, called *segments* that can be efficiently retrieved and updated. To manage these partitions we can use an *indirect index* that that allows retrieval of individual segments. In an indirect index

each dictionary entry points to a posting list of segments rather than the entire posting list of keyword locations. Using this design, queries need only retrieve the small segments that match a query rather than entire posting lists. Additionally, updates can read, modify, and write small sequential lists rather than very long lists.

8.6.3 Indexing Provenance and Relationships

Provenance data and generic relationships between files consist of both $\langle attribute, value \rangle$ pairs and a directed acyclic graph of relationships. All of the approaches discussed for storing metadata can be applied to storing $\langle attribute, value \rangle$ pairs for provenance and other relationships, so we need not design any special purpose indexing for those. The graphical structure of provenance and inter-file relationships, however, does require specific work on index creation, efficient graph-structured storage, and graph-aware search algorithms.

A provenance graph contains nodes both for persistent objects, such as files, as well as transient objects, such as processes and inter-process communication. An object's parents are the inputs used to produce that object and its children are items that derive from the object. As a concrete example, assume that process P reads files A and B, producing file C. The provenance graph that describes this execution has A and B as parents of P and C as a child of P. By transitivity, C is a descendent of both A and B.

The first important observation is that once an object is completely created, we know all of its parents, but not all of its children. This observation drives an ancestry-oriented organization, where nodes can be easily indexed through ancestor relationships, but not descendant relationships. We record parent relationships as named $\langle attribute, value \rangle$ pairs, where the attribute is the kind of parent (*e.g.*, an input file, a parent process, a container relationship) and the value identifies the parent.

The second observation is that queries on provenance relationships revolve around paths—sequences of nodes in the ancestry graph. For example, identifying all the files tainted by a buggy program is a query for the files in the subtree emanating from the buggy program.

Combining these two observations suggests that path and sub-trees are important to represent explicitly, but that sub-trees are problematic, because they require descendant relationships. Building upon the model used for storing other metadata discussed in Section 8.6.1, we can create descendant indexes lazily in each batch as well as path indexes that capture sets of ancestry relationships.

The second challenge to address is one of efficient graph storage. Conventional stores for provenance and other file relationships use one of a relational, XML, or RDF-based representation. XML representations are tree-based and do not generalize well to DAGs. Both relational and RDF-based representations store individual edges, so path queries become iterative or recursive queries along edges. While producing correct results, such queries are difficult for users to formulate and even more difficult to execute efficiently. The second version of the PASS system [74] uses a native $\langle key, value \rangle$ pair representation, using Berkeley DB [78]. While adequate, it too suffers from requiring iteration up (and down) the ancestry tree.

We will address this search problem through our path indexes. Path indexes use the statistical properties of the provenance graph to identify nodes with the greatest information content. Deriving from similar graphical analysis of the web [57, 58, 81], we can assign nodes in the provenance graph a slowly-changing *p*rovenance rank. Preliminary investigation in the context of PASS suggests that large changes in provenance rank as one traverses ancestry indicate a transition from a node with high information content to one with less information content. We can use these gaps in provenance rank to identify collections of nodes that should be stored together and indexed as a single path. Existing work on graph isomorphism can be used to identify patterns in the provenance graph which are equivalent, and condense it into a more compact representation.

Provenance and other file relationships will also be useful in determining clustering for our partitioned index. A single large relationship graph does not scale well to the billions of files in HEC file systems because it is too slow to query and too large to update efficiently. Instead, we will use the density of provenance relationships and other inter-file relationships to determine how to partition file metadata into the clusters

described in Section 8.6.1. By using relationship information to determine clustering, we will dramatically increase the likelihood that detailed search need only be done on a small fraction of the metadata clusters, greatly improving scalability. However, there are open questions for this approach that we must explore. First, we will investigate existing approaches for clustering in graphs [24,40,54,93], and evaluate their suitability for a provenance and relationship graph. Second, we must determine whether a file's metadata must be stored in a single cluster, or whether it may be replicated in multiple nearby clusters. Consistency is more difficult if a file is in multiple clusters; however, search performance may improve if a file can be stored in multiple clusters because a particular query may only reference one of the clusters in which the file resides. Third, there may be a point of diminishing returns as provenance linkages become more tenuous due to distance or age. Initial work on pruning provenance already exists, and we can further explore this research area to improve search resultt and performance.

Adding rich metadata and linking also opens up new horizons in file system search and personalization. Linking information can be used to identify context and find clusters of files [89], but it can also be used to identify users with similar interests and needs. Personalization can be achieved by tracking file dependencies and other relationships, and presenting the user related results, or by ranking results according to how "close" in the semantic graph they are to the user's files. We intend to explore the benefits of these and other novel personalization algorithms to present better results to individual users.

Our research in this area will therefore require four activities: analyzing provenance rank in large, HEC storage systems, developing techniques to use provenance rank and other relationships to construct efficient path indexes, designing and developing efficient storage structures for these path indexes, and using relationship information to partition the metadata into smaller, more manageable clusters.

8.7 Non-Hierarchical Name Space: Novel Approaches to Naming and Search

There is a clarion call for file systems that no longer rely on hierarchy to organize content. From Inversion [77] and LiFS [8] to iTunes [1] to Seltzer's position paper that "hierarchical file systems are dead" [88], systems are moving towards rich metadata and search for organizing and finding content. On such a file system, fast high quality search over metadata and content is more important than ever, since the hierarchy can no longer be used to navigate and find information. How well do these algorithms perform when the file system no longer has explicit hierarchy?

However, while high-performance, scalable metadata indexes are a key factor in facilitating better use of HEC file systems, providing powerful yet easy-to-use methods for accessing the indexes is just as critical. If users cannot easily access the file system, a high-efficiency indexing system is of little use. This section describes the research we will do to make our approach to file management both easy-to-use and sufficiently powerful to provide new functionality to HEC users.

Our solution must be easy-to-use, yet sufficiently expressive to allow advanced users or special-purpose software tools to provide advanced functionality such as complex searches for hard-to-find data and listings of files that must be regenerated because of a bug fix in a particular software package. Our approach to naming files must also be a superset of POSIX naming, since it is unrealistic to expect users to abruptly switch to a different naming convention. Instead, we will provide an approach that combines POSIX-style naming with more powerful, extensible naming schemes that facilitate dynamically-generated directories based on file attributes including metadata and provenance as well as per-user personalization. Our team already has some experience with languages for file naming [9], providing a strong foundation for this research.

8.7.1 Search-Oriented File Systems

A critical issue for managing files in HEC file systems is ensuring that users can easily find the files they need. However, existing systems use hierarchical file naming as their primary index, an approach whose limitations have engendered the creation of alternative parallel search systems [91] to help HEC users find

their data. We are taking a different approach by using a search-based interface as the *only* interface to the file system namespace. Our new architecture improves semantic file system performance and scalability by providing a combined indexing and storage layer on which semantic namespaces [8, 32, 36, 49] can be built. Thus, we must ensure that the interface is easy-to-use and has sufficient expressiveness to allow the construction of queries to pose complex but useful questions. We must also ensure that we present an interface that will allow users to continue using their familiar POSIX interface, gently migrating to our more flexible approach as need arises.

Because the scalable index and graph described in Section 8.6 is the only metadata structure in the file system, all references to files and "directories" are queries against the structure. There are several language approaches for constructing queries that include both file attributes and path information [8, 9, 47], and we will expand upon this work to develop an expressive query language that both users and applications can use to specify files and groups of files. Queries in this language can return zero files ("file not found"), exactly one file, or multiple files, which would correspond to a directory. Simply returning an unordered list of files would not be appropriate for a directory, which might contain thousands of files; instead, part of our research into path queries will explore the use of grouping, as is currently done in SQL. Our interface must also include a mechanism for queries such as "find files like this one." Designing a language that can provide such expressiveness without being too complex for simple uses is a key challenge for this research.

Web users are familiar with the problem of "information overload" in response to a search query; we will reduce this problem in our system by facilitating searches that are restricted to a local region of the provenance and relationship graph. This combination of file relationship information and per-file metadata has strong promise to greatly improve the quality of searches in a file system with 10^{18} files, so we will explore approaches that allow queries to include this information. Since relationships can themselves have $\langle attribute, value \rangle$ tuples, our language should allow queries to consider only particular links in establishing distance; this will allow queries to be restricted to files that are nearby or related from a particular user's point of view.

Including per-user personalization, context, and ownership information into the query is another challenge we will address. Web searches typically include little, if any, per-user context, yet such context is critical in finding appropriate files [89, 93]. Much of this information can be provided implicitly by having the file system monitor the files that have been used and including a digest of that information with each query sent to the file system. While file ownership information is straightforward to use, the decision of which file access information to gather and how to digest it is an open question that we will investigate.

By associating a ⟨POSIX_NAME, *filename*⟩ tuple with well-chosen inter-file relationships and creating null-content files to act as "directories," we can import an existing POSIX hierarchy into our system, allowing it to be used on already-extant file systems. This allows users and updated applications to take full advantage of flexible, dynamic search-based naming, while preserving POSIX functionality for legacy applications running on the same files.

8.7.2 New Functionality

By including both path-based information and per-file metadata in queries, our approach enables HEC users to find information that was previously either unavailable or difficult to obtain. To demonstrate the power and utility of the language that we will develop as part of this research, we will provide prototype queries that will immediately improve workflow for HEC users.

One area of immediate concern for HEC users is data provenance, yet prior efforts to manage provenance for scientific users have been limited and implemented as *ad hoc* additions on top of an existing file system [29, 39]. While it is important to rerun computational experiments in response to the discovery (and repair) of software bugs, it is impractical to rerun *every* such experiment. Our metadata index allows users to easily identify files that both need to be updated to reflect new software or other inputs and are widely used as inputs to other files. By constructing a tool to build a workflow to update the most critically needed

files that need regeneration, we will demonstrate the power of the system we design.

A second area of concern for HEC users is the *quality* of stored data. Data gathered by observation can be tagged with a quality metric corresponding to the trustworthiness of the data source. Derived data can then be assigned a quality metric that is a function of the quality of both its input data and the processing that was applied to it—some processes may be more trustworthy than others, perhaps because of better algorithms or more rigorously tested code. Again, we will build a tool to demonstrate the ability of our system to provide this information to HEC users.

UCSC is currently funded by NASA to work on a 1 year pilot project to provide terrestrial sensor information to earth scientists and biologists. We believe that this platform will be an excellent testbed for demonstrating the relevance of our research, albeit on a limited-size data set. Since it will be used by active HEC users, the system we are deploying can give us valuable insight into current usage patterns, the utility of the new functionality described above, as well as added features that would be useful for HEC users.

8.8 Related Work

To design a file system suitable for exascale computing, our research builds upon prior work in large-scale file systems metadata management and clustering, including content indexing. For HEC users to avoid wasted time spent adapting to the system, we enable interfaces more directly tailored to the HEC user by building upon semantic file systems concepts to provide dynamic naming and new file system search and management interfaces. Specifically we extend prior art in provenance, semantic file systems, as well as search personalization and security. We therefore discuss how our research relates to prior work in metadata management, indexing, provenance, and semantic file systems.

8.8.1 Metadata Management

The two fundamental index structures used in file system search are relational databases (DBMS) [23] and inverted indexes [41, 105]. Continual changes in technology and workload, and a continued reliance on traditional general-purpose DBMSs, have given rise to a belief that existing DBMS designs are not a "one size fits all" solution [17, 95, 96]. That is, a general-purpose DBMS cannot simply be tuned and calibrated to properly fit every workload. Many in the database community have argued and shown [44, 94] that using a traditional DBMS for a variety of search and indexing applications is often a poor solution and that a customized, application-specific design that considers the technology and workload requirements of the specific problem can often significantly outperform general-purpose DBMSs.

DBMSs were designed for business processing in the 1970s, not modern large-scale file system search [87]. DBMSs also have functionality, such as transactions and coarse locking, that are not needed for file system search and can add overhead [96]. Similarly, as evidenced by their contrasting designs, databases are often optimized for either read or write workloads and have difficulty doing both well [3, 45, 48]. Our proposed file system, designed specifically for HEC computing needs, would support both fast search performance and frequent real-time updates of the index.

Inverted indexes [41, 105] are designed as text databases and are the foundation of content search on the Internet and in current file system search. An inverted index, for a given text collection, builds a *dictionary* that contains a mapping for each of the K keywords in the collection to a list of the locations where the keywords occur in the file system. For large-scale file systems with many keywords, even with compression, the dictionary will be far too large to fit in a single machine's main memory [18]. Thus, they are often either stored on-disk or distributed across a cluster of machines.

While inverted indexes are the mainstay of modern text retrieval, they are not automatically suited for file system search. Small-scale systems make trade-offs between search and update performance, and are either too slow to handle real-time updates or offer unacceptable search performance. Large-scale designs used in web search applications are typically ineffective for file systems, as they require too much dedicated resources and need not rebuild their indexes as frequently. An HEC-tailored file system, such as the

one we will develop, would be be needed in order to effectively improve performance to levels acceptable for exascale HEC systems. Our proposed metadata management builds upon earlier work on the internal representation of metadata attributes in a file system [31,73].

8.8.2 Scalable Indexing

While there are file system search tools for both desktop [12,34,68,69] and enterprise [28,35,53,56] file systems, these tools are intended for smaller systems, on the order of tens of millions of files [33], and cannot easily scale to billions of files. Moreover, research has shown that their "one size fits all" approach to indexing is not effective and that systems need to consider their expected workloads in order to achieve the best performance [17,94,95]. Our investigation into metadata search [61] and current file system workloads [63] shows that significant benefits can be gained with a specialized solution.

We are not the first to look at how new indexing structures can improve performance or provide additional functionality. GLIMPSE [66] used a probabilistic inverted index and significantly reduced index space requirements by indexing in large blocks rather than storing the exact location of every keyword. Similarly, document-centric index pruning [20] reduces inverted index size by only indexing postings for the top K_D terms in a file based on a pseudo-relevance feedback step. Thus, the index size can be decreased by up to 80%, allowing a larger fraction to fit into main memory. Geometric partitioning [60] improves update performance, allowing an inverted index to handle continuous updates. Query-based partitioning [71] improves cost efficiency by partitioning the index based on term query frequencies and allowing postings lists for rarely searched terms to be placed on second-tier storage.

We will consider the above approaches, as well as other approaches to indexing content [101] and metadata in designing our metadata indexes. Our file system, like the Inversion file system [77], considers reorienting the file system design rather than adding additional features after the fact. Our approach, however, is not tied to an underlying DBMS, a fact that ultimately renders Inversion unsuitable as a scalable HEC file system.

Our research into techniques for partitioning the metadata index will build upon proven approaches for clustering that find patterns and similarities among items [40,97]. Document clustering has been proposed to automatically cluster web results [103] and to fit documents into an already existing set of categories [4]. Algorithms such as K-means and its variants can perform document clustering in time linear in the number of documents [65,104], making their performance well-suited for our purposes. We also will be examining clustering algorithms for graphs, such as the min-cut algorithm [24].

8.8.3 Provenance

Tracking, storing, and using provenance has been an active research topic for several decades within specialized domains. Systems like Chimera [29] and myGrid [39] track scientific data at the specific application level, while it has taken more recent efforts to explore the more general applicability of provenance information [22]. These efforts focused on databases and are not as general as file system provenance.

Tracking provenance at the file system level has many advantages, such as automatic provenance collection, a deeper view into full provenance that a domain specific system might miss, and potentially taking advantage of file system structure to aid in more efficient storage of provenance data. The Lineage File System [84] was one of the first file systems to track provenance information. In contrast to the Lineage File System, which used a MySQL database to store executables and command line histories, the Provenance Aware Storage System (PASS) [76] stores not only links to executable and other files, but also the hardware and software environment, a key piece when trying to recreate files or debug issues. Furthermore, PASS provided a layered architecture that facilitated integration of application-level and system-level provenance, increasing the value of both [74]. The PASS team also leveraged provenance to develop a versioning file system [75]. In conjunction with this, Holland *et al.* have designed a path-based query language that is particularly amenable to posing queries on the complex graphs that arise in provenance databases [47].

Auxiliary provenance security research has introduced methods for verifying and authenticating changes in content and ownership [42,43]. We will build upon this prior art to develop the first file system suitable for exascale computing that supplies fully-integrated high performance, secure provenance collection and indexing.

8.8.4 Semantic File Systems

Semantic file systems provide methods for accessing and managing file systems by replacing the traditional hierarchy with a namespace that can be dynamically created and searched using file attributes and relationships. The first semantic file systems [32,72] introduced the idea of foregoing the traditional hierarchical namespace and navigating the file system by means of "virtual directories" that are populated by specifying semantic attributes that each file in the directory must match. SFS [32] was built on top of a standard, unmodified UNIX file system, but could not offer real-time index consistency. Mogul [72] developed a prototype implementation that replaced the standard directory system with a searchable index; however, the system did not support personalization or content indexing, and was too slow to be the primary directory mechanism even in a relatively small file system; our goal is to *replace* the existing POSIX hierarchy with a more effective index.

Other systems, such as attribute-based naming [86], the Hierarchy and Content File System (HAC) [36], Presto [26], and LISFS [80], suggested a mixed model with both hierarchical and semantic attributes, but suffered from limitations such as dependence on a database back-end or limitations in scale. Haystack [52] proposed the use of RDF to organize information in the Web, providing both customized views and personalized search, but did not include relationships between URLs. The pStore system [102] also proposed storing semantic information in an RDF triple store, and used schemas to manage the different types of semantic information that might be available about files. pStore envisioned storage of causality information, as well as versioning, attributes, content semantics, and context. While our storage methodology is different, all of these types of information are valuable to capture, and will be leveraged in the system we develop. Our prior work on the Linking File system (LiFS) [7,8] was designed to take advantage of links between files, such those offered by provenance systems, as well as per-file metadata, to offer users rich navigation and search capabilities. Recent work [49] continues to explore semantic access to file systems, but does not consider provenance and does not offer scalability.

8.9 Preliminary Results

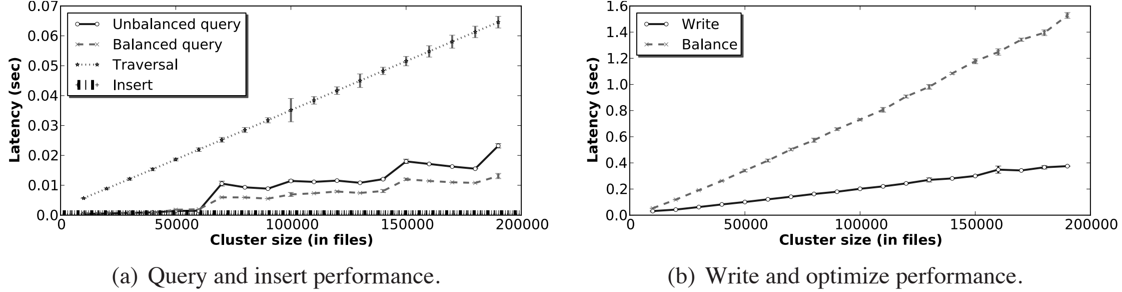
We present preliminary results for a prototype metadata management server that utilizes strategies discussed in Section 8.4. This prototype will serve as a basis for our integrated architecture as well as a testing platform for various algorithms. We also show preliminary results from testing different index partitioning methods, presenting a partitoning method which integrates file system security with partitioning the index. Early results show that the security aware partitioning method works quite well, indicating that some metadata attributes may be useful for index partitioning.

8.9.1 Metadata Management

We implemented our prototype as the metadata server (MDS) for the Ceph parallel file system [100]. In our prototype, each cluster has a maximum of 2,000 directories and a soft limit of 20,000 inodes, keeping them fast to access and query. We discuss the reasoning behind these numbers later in this section. The K-D tree in each cluster is implemented using `libkdtree++` [64], version 0.7.0. Each inode has eleven attributes that are indexed, listed in Table 1. Each Bloom filter is about 2 KB in size—small enough to represent many attribute values while not using significant amounts of memory. The hashing functions we use for the file size and time attributes allow bits to correspond to ranges of values. Each cluster's metadata cache is 100 KB in size. While our prototype implements most metadata server functionality, there are a number of features not yet implemented. Among these are hard or symbolic links, handling of client cache leases, and

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

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



(a) Query and insert performance.

(b) Write and optimize performance.

Figure 3: ***Cluster indexing performance.*** *Figure 3(a) shows the latencies for balanced and unbalanced K-D tree queries, brute force traversal, and inserts as cluster size increases. A balanced K-D tree is the fastest to search and inserts are fast even in larger clusters. Figure 3(b) shows latencies for K-D tree rebalancing and disk writes. Rebalancing is slower because it is requires $O(N \times \log N)$ time.*

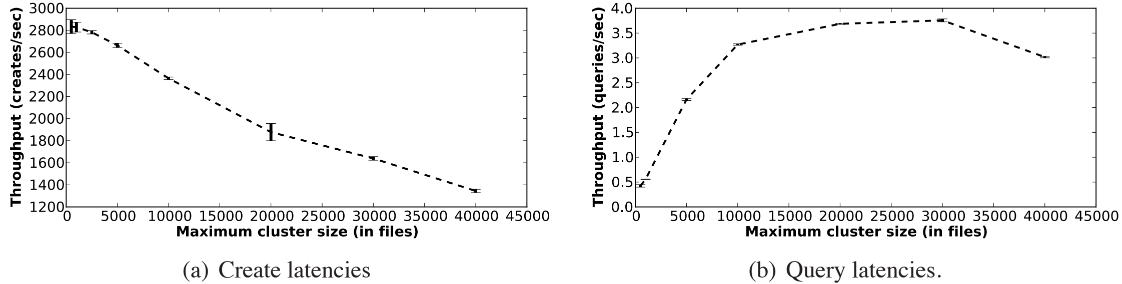
metadata replication. None of present a significant implementation barrier, and none significantly impact performance; we will implement them in the future.

Cluster Indexing Performance. We evaluated the update and query performance for a *single* cluster in order to understand how indexing metadata in a K-D tree affects performance. Figure 3(a) shows the latencies for creating and querying files in a single cluster as the cluster size increases. Results are averaged over five runs with the standard deviations shown. We randomly generated files because different file systems have different attribute distributions that can make the K-D tree un-balanced and bias results in different ways [5]. We used range queries for between two and five attributes.

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

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

In contrast to query performance, insert performance remains fast as cluster size increases. The insert algorithm is similar to the exact match query algorithm, requiring only $O(\log N)$ time to complete. Even for larger K-D trees, inserts take less than 10 us. The downside is that each insert makes the tree less balanced,



(a) Create latencies (b) Query latencies.

Figure 4: ***Metadata clustering.*** *Figure 4(a) shows create throughput as maximum cluster size increases. Performance decreases with cluster size because inode caching and Bloom filters become less effective and K-D tree operations become slower. Figure 4(b) shows that query performance is worse for small and large sizes.*

degrading performance for subsequent queries until the tree is rebalanced. Thus, while inserts are fast, there is a hidden cost being paid in slower queries and having to rebalance the tree later.

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

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

Figure 4(a) shows the total throughput for creating 500,000 files from the Web trace over five runs as the maximum cluster size varies from 500 to 40,000 inodes. As the figure shows, Create throughput steadily decreases as maximum cluster size increases. While the throughput at cluster size 500 is around 2,800 creates per second, at cluster size 40,000, which is an $80\times$ increase, throughput drops roughly 50%. Disk utilization is not the issue, since both use mostly sequential disk writes; rather, the decrease is primarily due to having to operate on larger K-D trees. Smaller clusters have more effective metadata caching (less data to cache per K-D tree) and Bloom filters (fewer files yielding fewer false positives). Additionally, queries on smaller K-D trees are faster. Since journal writes and K-D tree insert performance do not improve with cluster size, a larger maximum cluster size has little positive impact.

Figure 4(b) shows that query performance scales quite differently from create performance. We used a simple query that represented a user search for a file she owns with a particular name (*e.g.* filename equal to `mypaper.pdf` and owner id equal to `3704`). We find that query throughput *increases* $7-8\times$ as maximum cluster size varies from 500 to 25,000. When clusters are small, metadata clustering is not as helpful because many disk seeks may still be needed to read the metadata needed. As clusters get larger disk utilization improves. However, throughput decreases 15% when maximum cluster size increases from 30,000 to 40,000 files. When clusters are too large, time is wasted reading unneeded metadata, which can also displace useful information in the cluster cache. In addition, larger K-D trees are slower to query. The sweet spot seems to be around 20,000 files per cluster, which we use as our prototype's default.

8.9.2 Scalable Indexing: Security Aware Partitioning

For search to be secure, a file system needs to be aware of a combination of emphread, *write*, and *execute* permissions for all security levels. For a user on a UNIX system to access files or subdirectories within a directory, the directory's *execute* bit must be set for some role which the user fulfills. If the *other* execute bit is set, any user not the owner or a member of the group can access files in or below that directory. Otherwise the user must be the file's owner or a member of the group, and have the corresponding *execute* bit set. Further, the *execute* bit must be set on every directory preceding the current one in the path. In other words, access is determined by the logical AND of the access permissions of every directory in a file's path, relative to a specific user's roles.

However, for a user to actually view the contents of a directory, they must not only have *execute* permissions, but *read* permissions as well. Unlike *execute*, *read* does not require permissions all along the path. It is possible to read the contents of a directory as long as the user has *read* permissions on the last directory in the path. This allows security operations such as permitting users to own a directory without being able to list the contents of directories above that one. For instance, a system may choose to not let users view which other users have directories in `/home`, even though the users themselves are the owners of `/home/`<`username`> directories.

Therefore, Security Aware Partitioning partitions the file system according to *group* and *user* security permissions. The algorithm walks the file system in a breadth first search. Access permission is determined by examining all permissions in the directories above the file or directory in question. If the permissions on the current file or directory are more restrictive than that of the current partition or the *user* or *group* has changed, then a new partition is created. Subfolders and files are added until another restriction in permissions is encountered. This ensures that all files and directories in a given partition share the same permissions.

In order to evaluate the effectiveness of Security Aware Partitioning, we compared it to the following partitioning algorithms: a greedy time based algorithm, an interval time based algorithm, a user based algorithm, Cosine correlation clustering, Cosine correlation clustering with Latent Semantic Analysis (LSA) – SmartStore [50], and a greedy depth first search partitioning algorithm – Spyglass [61]. We ran each alogrithm over a crawl containing file server metadata and evaluated the resulting partitions using the following criteria: size of the partitions, runtime and memory usage (evaluated using Big-O run times in order to account for variations in the algorithms), the actual files within the partitions, partition entropy, and information gain. Partition entropy measures the variance of attribute values in a given partition, and was calculated using the Shannon formula for information [90]. Information gain measures the amount of information you gain about an attribute by being in a given partition, and was calculated using Quinlan's description [82]. These criteria were selected because it allows the partitioning algorithms to be compared without building complete systems with working implementations of each algorithm.

Security Aware Partitioning is linear with respect to the size of the data set. Files and directories do not need to be compared to other files/directories, just to the security permissions of the current partition. Because UNIX style permissions rely on the hierarchy, we do a recursive tree descent. The permissions of each file in the path are stored on the stack, for a memory requirement of $log(n)$, where n is the number of files. The wide branching factor of directories makes the constants quite low in practice. This memory usage is a constraint specific to UNIX and would not necessarily apply to other architectures.

Partition sizes directly impact the effectiveness of search. If the partitioning algorithm results in a large number of small partitions, then finding relevant files may result in many calls to the disk in order to load all of the required indexes. By contrast, if partitions are too large then the index cannot easily fit into memory. An ideal partitioning algorithm will result in most partitions being near the maximum size, with a fairly low variance in size. We show the mean, median, and variance for partition size in Table 2. With the exception of the greedy algorithms, which use a fixed number of files per partition, all of the algorithms had a large

Table 2: *SOE Size Statistics*

	Greedy DFS	Greedy Time	Interval	User	Security	Cosine	LSA
Number of partitions	81	64	8	384	29479	131	1370
Mean size	85203	107835	862683	17973	234	5037	52683
Standard Deviation	44487	43634	2163134	128252	3309	36776	292193

number of smaller partitions and a few large partitions. This is not unexpected. The skewed nature of metadata has been explored by Leung [61], and any algorithm which relies on it is likely to be somewhat skewed in distribution. This means any non-greedy algorithm will construct many smaller indexes that will later need to be accessed. However, this cost can potentially be mitigated through clever on-disk layout and data structures. Non-greedy algorithms also result in partitions which are over 100,000 files and may be too large for indexing. In this case, a secondary criterion (such as time), could be used to split the partition into more manageable sub-indexes.

Security Aware Partitioning has a low standard deviation, suggesting that partitions tend to be approximately the same size. However, the mean size is at least an order of magnitude lower than any of the other algorithms. This means Security Aware Partitioning creates a large number of small partitions. Since the partitioning is based on hierarchy, this algorithm might benefit from merging partitions across hierarchical boundaries if they have a matching set of users who can access them. We plan to investigate strategies for merging partitions.

Partition entropy measures the "goodness" of a partition, by measuring the entropy per attribute within each partition. This measures the number of values of an attribute in a given partition. A low entropy suggests that the attribute values within that partition are somewhat homogeneous – there are only a few attribute values in that partition. For entropy calculations, we did not include the path name or the inode number, since these will almost always be unique to a specific file or directory. In Figure 5 we present the cumulative distribution function of entropy for different attributes, with each algorithm displayed. Here, a fast growth rate implies that most of the entropy for that algorithm was low, and therefore the algorithm will be more efficient at retrieving data related to that attribute. We have selected a few attributes from the SOE data to display, based on common user queries: specifically file type and user id.

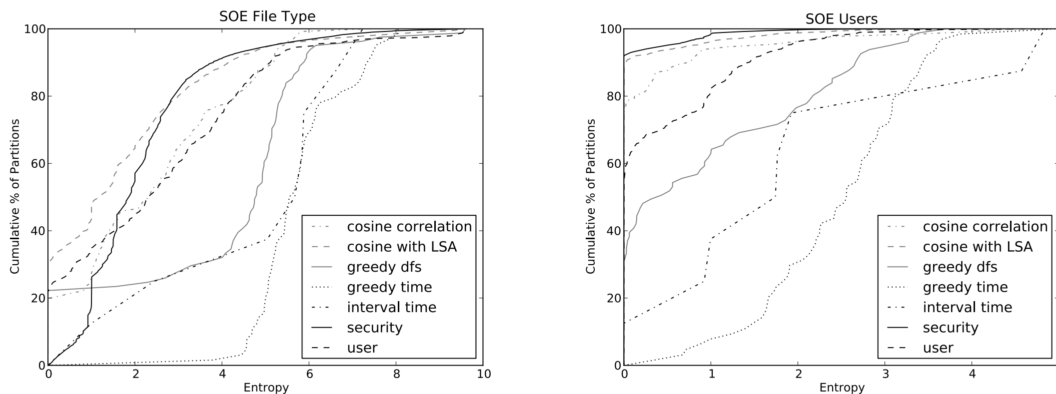


Figure 5: *CDFs of entropy by mode, mtime, type, and uid for percentage of partitions. Algorithms which grow more quickly in this graph are better for search. Note that the security algorithm grows quickly, meaning has excellent entropy for all attributes.*

8.10 Conclusion

For HEC and scientific computing applications there is an urgent and growing need for improved data management. Ultimately, the goal is to reduce the amount of time and effort spent by scientists and similar HEC users managing and moving their data. To enable more powerful and effective data workflow management we will provide a novel file system design. Our approach allows the automation of rich metadata and provenance collection within a general purpose filesystem. The choice to implement a new file system allows us to provide a data management solution that can accommodate legacy data while offering a more adaptable and accommodating interface to the data.

8.11 Project Timetable

Year 1. Research goals for the first year include:

- Test the scalability of existing methods for dynamically extracting and validating metadata, content-based metadata, and provenance, and investigate new approaches for better performance in exascale systems.
- Develop algorithms for automatically generating statistics about the system including metadata and provenance information, and the impact having these statistics has on indexing.
- Further evaluating the needs of HEC users and the specific workflow challenges in HEC systems and expanding our interactions with the scientific computing communities by working with researchers at HEC facilities.
- Explore algorithms for graph homomorphism to evaluate suitability for file systems and scalability to billions of nodes.
- Develop designs for large-scale indexes that leverage non-volatile memories for both indexing and fast updates.
- Explore new hardware technologies as the basis for an exascale system and identify the challenges in implementing and optimizing the integrated metadata server on such hardware.

Year 2. Research goals for the second year include:

- Integrate dynamic extraction into a file system containing a partition-based metadata server with provenance and content-based metadata.
- Refine a statistics generation algorithm for exascale systems and integrate it into the file system.
- Implement experimental prototypes, and develop initial filesystem design.
- Continue the investigation and development of new approaches for improving metadata management in exascale systems.
- Continue developing automatic statistics generation and gathering algorithms.
- Continue evaluation of new storage technologies and integration into the filesystem design.
- Further evaluate the needs of HEC users and evaluate workflow and interface elements.

Year 3. Research goals for the third year include:

- Prototype testing and further development, including evaluation of partitioning.
- Experimental evaluation of hierarchical indexes and related techniques to improve scalability of searching across more than 10^6 partitions.
- Conduct further experiments on the metadata server to demonstrate scalability and to determine an optimal partitioning scheme.
- Develop an integrated search algorithm that uses the system statistics, metadata, and provenance to provide efficient and accurate results, and continued evaluation of its effectiveness for HEC users.
- Continue evaluation of new storage technologies and their integration into the filesystem design and prototype.