Copernicus: A Scalable, High-Performance Semantic File System

Technical Report UCSC-SSRC-09-06
October 2009

Andrew W. Leung Aleatha Parker-Wood Ethan L. Miller
aleung@cs.ucsc.edu aleatha@cs.ucsc.edu elm@cs.ucsc.edu

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

Copernicus: A Scalable, High-Performance Semantic File System

Andrew W. Leung Aleatha Parker-Wood Ethan L. Miller
University of California, Santa Cruz

Abstract

Hierarchical file systems do not effectively meet the needs of users at the petabyte-scale. Users need dynamic, search-based file access in order to properly manage and use their growing sea of data. This paper presents the design of Copernicus, a new scalable, semantic file system that provides a searchable namespace for billions of files. Instead of augmenting a traditional file system with a search index, Copernicus uses a dynamic, graph-based file system design that indexes file attributes and relationships to provide scalable search and navigation of files.

1 Introduction

Today, file systems store petabytes of data across billions of files and can serve thousands of users. Current hierarchical file organizations, which are over 40 years old and are meant for orders of magnitude fewer files [3], no longer match how users access and manage their files [14]. As a result, users are no longer able to easily know *where* to find their data and must use lengthy brute force search when they do not know exactly where it is.

Instead, users need to be able to describe *what* they are looking for. For example, a scientist's HPC application may generate thousands of files containing data from experiments; finding the few files with interesting results or those with related or similar results can be difficult. Using and sharing these files requires that files be accessible via their results (*i.e.,* content), the experiment parameters (*i.e.,* metadata), and the files and data used to generate the results (*i.e.,* inter-file relationships). Users are hard-pressed to address even common problems, such as locating the files that consume the most disk space or finding where files for an application have been installed.

As system capacities have grown, this problem has been addressed by augmenting file systems with separate applications that provide search and indexing functionality. Search applications, which use additional file metadata and content indexes for faster search, have become popular on both desktop [1] and enterprise [7] file systems. However, these applications are simply makeshift solutions to a more fundamental problem: hierarchical file systems do not provide the search and management capabilities that users require. Additionally, storing files in both the file system and a search application introduces space, performance, and usability overheads that can limit their effectiveness at the petabyte-scale.

This approach is far from ideal and suggest that file systems themselves be re-designed to provide the functionality required by users at the petabyte-scale. This paper makes the following contributions: (1) it argues that search and a semantic namespace should be primary goals of the file system; (2) it presents some basic requirements and challenges for building a solution; (3) and presents the design of Copernicus, a file system that aims to address these challenges at large scales. The core of the Copernicus file system is a dynamic graph-based index that clusters semantically related files into vertexes and allows inter-file relationships to form edges between them. This graph replaces the traditional directory hierarchy, can be efficiently queried, and allows the construction of dynamic namespaces. The namespace allows "virtual" directories that correspond to a query and navigation using inter-file relationships. Additionally, by integrating search directly into the file system Copernicus can effectively scale to billions of files.

1.1 Motivating Examples

The following examples show how Copernicus can improve file management.

Understanding file dependencies. Consider a scientist running an HPC DNA sequencing application. To interpret the results, it is useful to know how the data is being generated. As the experiment runs, Copernicus allows the results to be searched in real time. If a compelling result is found, a virtual directory can be created containing files from past experiments with similar results. By searching the provenance links of those files, the scientist can find which DNA sequencing libraries or input parameters are the common factor for all of the result files.

System administration. Imagine a storage administrator who discovers a serious bug in a script that has affected an unknown number of files. To locate and fix these files, the administrator can search provenance relationships to find the contaminated files (*e.g.,* files opened by the script) and build a virtual directory containing these files. A cor-

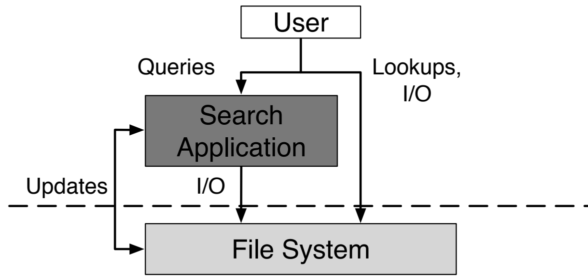

Figure 1: ***File search applications.*** *The search application resides on top of the file system and stores file metadata and content in separate search-optimized indexes. Maintaining several large index structures can add significant space and time overheads.*

rected version of the script can be run over the files in this directory to quickly undo the erroneous changes.

Finding misplaced files. Consider a user working on a paper about file system search and looking for related work. The user recalls reading an interesting paper while working on "motivation.tex" but does not know the paper's title or author. However, using temporal links and metadata, a virtual directory can be constructed of all files that were accessed at the same time as "motivation.tex", are PDFs, and contain "file system" and "search". The directory allows the user to easily browse the results.

2 Petabyte-Scale Challenges

As file systems have grown to store many billions of files, search-based access and navigation have become critical requirements. Unfortunately, current file system and search solutions are not well equipped to address these challenges.

2.1 Current Limitations

Hierarchical file organization was designed for systems with thousands of files and aimed to provide only basic navigation for persistent storage [3]. As a result, basic hierarchical organization has several limitations that do not match the needs of users who use and manage billions of files. First, files are only allowed to have a single categorization (*i.e.,* its pathname). Useful information describing a file cannot be used to access it and is often lost; it is often impossible for users to recall a single categorization in large systems. Second, files can only be related through *parent* → *child* relationships. Other important inter-file relationships, such as provenance, temporal context, or use in a related project, are lost. Third, hierarchies provide no convenient search functionality. When a file's location is not known, a brute force search, which can be extremely slow, is required.

The limitations of current hierarchical organization are well documented, and there are a variety of proposed solutions. Early solutions, such as `find` and `grep`, aimed to make brute force search less cumbersome. Semantic file systems [6] provided new namespaces that allowed search-based access to files and construction of "virtual" directories that were associated with queries. Semantic file systems provided better methods for accessing files, but were designed as applications above the file system which caused serious performance and consistency problems. Other search applications improved performance by using new index designs tailored for file systems [10, 12]. Today, tools that provide metadata and content search for desktop [1] and small-scale (tens of millions of files) enterprise [7] file systems are common.

However, current solutions are applications that reside *separately* from the file system. As a result, they can simply *conceal* current hierarchical limitations, rather than solve them. Figure 1 shows how these applications interact with the file system. The search application maintains search indexes for file metadata and content, such as databases or inverted files, which are stored persistently as files in the file system.

Separate search applications are not an ideal solution because they require a level of indirection that is inefficient. A resource overhead is incurred, which can be significant at the petabyte-scale, because each file requires resources in both the file system's and application's indexes. Additionally, the application must track file system changes, either by crawling or monitoring activity, a slow process that often leaves the application inconsistent with the file system because each file modification requires updates to the indexes of each. Moreover, querying the search application can be highly inefficient. A query requires the search indexes to be accessed, which must then leverage the file system's index (since they are stored in the file system); actually retrieving the file requires another look up in the file system index. These kinds of inefficiencies are well known to the database community [17] and the reason that many databases manage their own storage. Finally, users must interact with multiple interfaces depending on how they want to access their files.

We posit that a scalable, searchable namespace *is* functionality that file systems should provide because 1) it is becoming increasingly important functionality, 2) a separate application limits scalability and usability and 3) the file system already provides existing indexing functionality that can be leveraged. Other recent work supports the idea that "hierarchical file systems are dead" [14].

2.2 Modern File System Requirements

To address the needs of today's users, modern large-scale file systems must meet some basic requirements.

Flexible naming. The main drawback with current hierarchies is their inability to allow flexible and semantic access to files. Files should be able to be accessed using their attributes *and* relationships. Thus, the file system must efficiently extract and infer the necessary attributes and relationships and index them in real-time.

Dynamic navigation. While search is extremely useful for retrieval, users still need a way to navigate the namespace. Navigation should be more expressive than just *parent* → *child* hierarchies, should allow dynamically changing (or virtual) directories and need not be acyclic. Relationships should be allowed between two files, rather than only directories and files.

Scalability. Large file systems are the most difficult to manage, making it critical that both search and I/O performance scale to billions of files. Effective scalability requires fine-grained control of file index structures that allow disk layouts and memory utilization to properly match workloads.

Backwards compatibility. Existing applications rely on hierarchical namespaces. It is critical that new file systems be able to support legacy applications to facilitate migration to a new paradigm.

3 Copernicus Architecture

Copernicus is designed as an object-based parallel file system so that it can achieve high scalability by decoupling the metadata and data paths and allowing parallel access to storage devices [18]. However, Copernicus's techniques are applicable to a wide range of architectures. Object-based file systems consist of three main components: clients, a metadata server cluster (MDS), and a cluster of object-based storage devices (OSD). Clients perform file I/O directly with OSDs, but submit metadata and search requests to the MDS, which manages the namespace; thus, most of the Copernicus design is focused on the MDS.

Copernicus achieves a scalable, semantic namespace using several new techniques. A dynamic graph-based index provides file metadata and attribute layouts that enable scalable search, as shown in Figure 2. Files that are semantically similar and likely to be accessed together are grouped into *clusters*, which are similar to traditional directories, and form the vertices of the graph. Inter-file relationships, such as provenance [13, 15] and temporal access patterns [16], create edges between files that enable semantic navigation. Directories are "virtual," and are instantiated by queries. Backwards naming compatibility can be enabled by creating a hierarchical tree from the graph. Clusters store metadata and attributes in search-optimized index structures. The use of search indexes for native storage mechanisms allows Copernicus to be easily searched without additional search applications. Finally, a new journaling mechanism allows file metadata modifications to be written quickly and safely to disk while still providing real-time index updates.

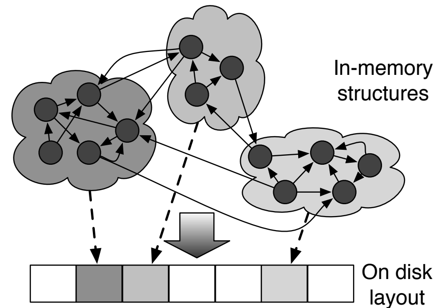

Figure 2: ***Copernicus overview.*** *Clusters, shown in different colors, group semantically related files. Files within a cluster form a smaller graph based on how the files are related. These links, and the links between clusters, create the Copernicus namespace. Each cluster is relatively small and is stored in a sequential region on disk for fast access.*

3.1 Graph Construction

Copernicus uses a graph-based index to provide a metadata and attribute layout that can be efficiently searched. The graph is managed by the MDS. Each file is represented with an inode and is uniquely identified by its inode number. Inodes and associated attributes—content keywords and relationships—are grouped into physical clusters based on their semantic similarity. Clusters are like directories in that they represent a physical grouping of related files likely to be accessed together, in the same way that file systems try to keep files adjacent to their containing directory on disk. This grouping provides a flexible, fine-grained way to access and control files. However, unlike directories, cluster groupings are semantic rather than hierarchical and are transparent to users: clusters only provide physical organization for inodes. Given a file's inode number, a pseudo-random placement algorithm, CRUSH [19], identifies the locations of the file's data on the OSDs, meaning data pointers are not stored within the inode.

Inodes are grouped into clusters using *clustering policies*, which define their semantic similarity. Clustering policies may be set by users, administrators, or Copernicus, and can change over time, allowing layouts to adjust to the current access patterns. Inodes may move between clusters as their attributes change. Example clustering policies include clustering files for a common project (*e.g.,* files related to an HPC experiment), grouping files with shared attributes (*e.g.,* files owned by Andrew or all virtual machine images), or clustering files with common access patterns (*e.g.,* files often accessed in sequence or in

parallel). Previous work has used Latent Semantic Indexing (LSI) as a policy to group related files [8]. In Copernicus, files are allowed to reside in only one cluster because maintaining multiple active replicas makes synchronization difficult. Clusters are kept relatively small, around 10^5 files, to ensure fast access to any one cluster; thus, a large file system may have 10^4 or more clusters.

Copernicus creates a namespace using the semantic relationships that exist between files. Relationship links are created implicitly by Copernicus depending on how files are used and can also be created explicitly by users and applications. Unlike a traditional file system, links only exist between two files; directories in Copernicus are "virtual" and simply represent the set of files matching a search query. Virtual directories allow dynamic namespaces to be constructed, while links provide an easy, semantic way to navigate the namespace. Relationships are directed and are represented as triples of the form $\langle$*relationship type*, *source file*, *target file*$\rangle$, and can define any kind of relationship. Relationship links may exist between files within the same or different clusters as illustrated in Figure 2. The graph need not be acyclic, permitting more flexible relationships.

3.2 Cluster Indexing

Files in Copernicus may be retrieved using their metadata, content, or relationship attributes. Each cluster stores these attributes in separate search-optimized index structures, improving efficiency by allowing files to easily be searched without a separate application. File metadata is represented as $\langle attribute, value \rangle$ pairs and includes simple POSIX metadata and extended attributes. Metadata is indexed in a in-memory, multi-dimensional binary search tree called a K-D tree [2]. K-D trees, which have been used previously to index metadata [10], provide fast, logarithmic point, range, and nearest neighbor queries. A key advantage of multi-dimensional search trees is that all metadata attributes can be indexed in a single structure, as opposed to a B-tree, which would require one per attribute. Since clusters are relatively small, each K-D tree can often be stored in a sequential region on disk. This layout, which is similar to embedded inodes [5], provides fast read access and prefetching of related metadata.

Relationship attributes are also stored in a K-D tree. K-D trees allow any combination of the relationship triple to be queried. If a relationship exists between files in different clusters, the cluster storing the source file's inode indexes the relationship, to prevent duplication. This K-D tree can also usually be stored sequentially on disk.

Each cluster stores full-text keywords, which are extracted from its files' contents using application-specific *transducers*, in its own inverted index. This design allows keyword search at the granularity of clusters and helps keep posting lists small so that they can be kept sequential on disk. A global indirect index [11] is used to identify which clusters contain posting lists for a keyword. An indirect index consists of a keyword dictionary with each keyword entry pointing to a list of $\langle cluster, weight \rangle$ pairs, allowing the MDS to quickly identify the clusters most likely to contain an answer to a query and rule out those clusters that *cannot* satisfy the query.

3.3 Query Execution

All file accesses (*e.g.,* `open()` and `stat()`) translate to queries over the Copernicus graph index. While navigation can be done using graph traversal algorithms, queries must also be able to identify the clusters containing files relevant to the search. Since semantically related files are clustered in the namespace, it is very likely that the vast majority of clusters do not need to be searched. This has already been shown to be the case in hierarchical file systems [10], despite only modest semantic clustering. Additionally, Copernicus employs an LRU-based caching algorithm to ensure that queries for hot or popular clusters do not go to disk.

For file metadata and relationships, Copernicus identifies relevant clusters using *signature files* [4]—bit arrays with associated hashing functions that compactly describe the contents of a cluster. When a cluster stores a metadata or relationship attribute, it hashes its value to a bit position in a signature file, which is then set to one. To determine if a cluster contains any files related to a query, the values in the query are also hashed to bit positions, which are then tested. If, and only if, all tested bits are set to one is the cluster read from disk and searched.

Signature files are one-dimensional; thus, one is maintained for each type of attribute indexed. To ensure fast access, signature files are kept in memory. To do this, each is kept small: 10^3 to 10^5 bit positions per signature file. While false positives can occur when two values hash to the same bit position, the only effect is that a cluster is searched when it does not contain files relevant to the query, degrading search performance but not impacting accuracy.

The sheer number of possible keywords occurring in file content make signature files ineffective for keyword search. However, the indirect index allows fast identification of the clusters containing posting lists for the query keywords. For each keyword in the query, the list of clusters containing the keyword is retrieved. Assuming Boolean search, the lists are then intersected, producing the set of clusters that appeared in all lists. Only the posting lists from the clusters appearing in this set are retrieved and searched. The weights can be used to further optimize query processing, first searching in clusters that are most likely to contain the desired results.

3.4 Managing Updates

Copernicus must effectively balance search and update performance, provide real-time index updates, and provide the data safety that users expect. Copernicus uses a journal-based approach for managing metadata and relationship updates, and a client-based approach for managing content keywords. When file metadata or relationships are created, removed or modified, the update is first written safely to a journal on disk. By first journaling updates safely to disk, Copernicus is able to provide needed data safety in case of a crash. The K-D tree containing the file's inode or relationship information is then modified and marked as dirty in the cache, thereby reflecting changes in the index in real-time. When a cluster is evicted from the cache, the entire K-D tree is written sequentially to disk and its entries are removed from the journal. Copernicus allows the journal to grow up to hundreds of megabytes before it is trimmed, which helps to amortize multiple updates into a single disk write.

Unfortunately, K-D trees do not efficiently handle frequent inserts and modifications. Inserting new inodes into the tree can cause it to become unbalanced, degrading search performance. As a result, K-D trees are rebalanced before they are written to disk. Also, inode modifications first require the original inode to be removed and then a new inode to be inserted. Both of these operations are fast compared to writing to the journal, but since disk speed dictates update performance, storing the journal in NVRAM can significantly boost performance.

Clients write file data directly to OSDs. When a file is closed, Copernicus accesses the file's data from the OSDs and use a transducer to extract keywords. To aid this process, clients submit a list of write offsets and lengths to the MDS when they close a file. These offsets tell the MDS which parts of the file to analyze and can greatly improve performance for large files. Cluster posting lists are then updated with extracted keywords. Since cluster posting lists are small, an in-place update method [9] can be used, ensuring that they remain sequential on disk.

4 Conclusions and Open Questions

Hierarchical file organization was designed for the systems of yesterday. At the petabyte-scale, file systems must break away from this paradigm and provide a semantic, searchable namespace where users can ask for *what* they want, rather than saying *where* to find it. Existing search applications, which are separate from the file system, will not effectively scale. To address this problem we designed Copernicus, which uses a novel graph-based index to provide a semantic namespace and scalable performance.

Because Copernicus changes many common file system concepts, a number of practical questions remain open. First, how effectively does a generic graph index scale and how effective are search trees at handling file system workloads? Second, what are the challenges with providing needed functionality such as security? Third, can the Copernicus graph be leveraged for better file search result ranking or interface? We intend to explore these and other issues as we continue our design.

References

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

[2] BENTLEY, J. L. Multidimensional binary search trees used for associative searching. *CACM 18*, 9 (Sept. 1975).

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

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

[5] GANGER, G. R., AND KAASHOEK, M. F. Embedded inodes and explicit groupings: Exploiting disk bandwidth for small files. In *USENIX 1997*.

[6] GIFFORD, D. K., JOUVELOT, P., SHELDON, M. A., AND O'TOOLE, JR., J. W. Semantic file systems. In *SOSP 1991*.

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

[8] HUA, Y., JIANG, H., ZHU, Y., FENG, D., AND TIAN, L. SmartStore: A new metadata organization paradigm with metadata semantic-awareness for next-generation file systems. In *Proceedings of SC '09*.

[9] LESTER, N., MOFFAT, A., AND ZOBEL, J. Efficient online index construction for text databases. *ACM ToDS 33*, 2 (2008).

[10] LEUNG, A., SHAO, M., BISSON, T., PASUPATHY, S., AND MILLER, E. L. Spyglass: Fast, scalable metadata search for large-scale storage systems. In *FAST 2009*.

[11] LEUNG, A. W., AND MILLER, E. L. Scalable full-text search for petascale file systems. In *PDSW 2008*.

[12] MANBER, U., AND WU, S. GLIMPSE: A tool to search through entire file systems. In *USENIX Winter 1994*.

[13] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. I. Provenance-aware storage systems. In *USENIX 2006*.

[14] SELTZER, M., AND MURPHY, N. Hierarchical file systems are dead. In *HotOS-XII*.

[15] SHAH, S., SOULES, C. A. N., GANGER, G. R., AND NOBLE, B. D. Using provenance to aid in personal file search. In *USENIX 2007*.

[16] SOULES, C. A. N., AND GANGER, G. R. Connections: using context to enhance file search. In *SOSP 2005*.

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

[18] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *OSDI '06*.

[19] WEIL, S. A., BRANDT, S. A., MILLER, E. L., AND MALTZAHN, C. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *SC '06*.