GUFI: Fast, Secure File System Metadata Search for Both Privileged and Unprivileged Users

Dominic Manno* Jason Lee* Prajwal Challa† Qing Zheng* David Bonnie* Gary Grider* Bradley Settlemyer‡

*Los Alamos National Laboratory
Los Alamos, NM, USA
{dmanno,jlee,qzheng,dbonnie,ggrider}@lanl.gov

†University of Texas at Arlington
Arlington, TX, USA
{vxc5208}@mavs.uta.edu

‡NVIDIA
Austin, TX, USA
{bsettlemyer}@nvidia.com

***Abstract*—Modern High-Performance Computing (HPC) data centers routinely store massive data sets resulting in millions of directories and billions of files. To efficiently search and sift through these files and directories we present the Grand Unified File Index (GUFI), a novel file system metadata index that enables both privileged and regular users to rapidly locate and characterize data sets of interest. GUFI uses a hierarchical index that preserves file access permissions such that the index can be securely accessed by users while still enabling efficient, advanced analysis of storage system usage by cluster administrators. Compared with the current state-of-the-art indexing for file system metadata, GUFI is able to provide speedups of 1.5× to 230× for queries executed by administrators on a real production file system namespace. Queries executed by users, which typically cannot rely on cluster-wide indexing, see even greater speedups using GUFI.**

***Index Terms*—Parallel query processing, file system metadata management, file system access control, index sharding**

I. Introduction

High-performance Computing (HPC) data centers provide a wide variety of data services to meet the demands of diverse workloads [1, 2]. Specific storage systems are tailored to provide space for specific services — such as user specific files ("/home"), collaboration spaces ("/project"), bulk data storage services ("/scratch"), and archival storage services ("/archive") — such that each service features a separate namespace at a separate mount point. Storage systems tailored to workloads are able to provide higher performance than a general-purpose system [3]. To direct data to the right system, workflows are constructed to orchestrate data movement between different data services during each data processing step [4]. While the specialized nature of HPC storage enables users to efficiently execute diverse scientific workflows, it also creates a burden on system users and cluster administrators to efficiently locate and manage the large amount of files spread across a wide range of available storage systems.

When the total file count is small, cluster-wide data management tasks may be efficiently accomplished by standard data management tools such as *du* and *find*. This is because most data storage systems today provide a hierarchical namespace that supports the traditional POSIX file system interface such that a large amount of common data management queries can be directly decomposed into individual directory scan and file attribute retrieval requests to underlying storage systems' metadata managers. Over time, irrespective of cluster size, the files created by each data service grow to huge numbers and capacities [5]. Modern HPC data centers routinely store massive data sets resulting in millions of directories and billions of files [6]. Even though today's fastest distributed file system metadata servers are capable of hundreds of thousands of — or even millions of — metadata operations per second [7–10], the large amount of small metadata query requests that must be sent to individual storage systems makes high-level data management queries increasingly costly, leading to prohibitive query response times. This is especially so when queries require scanning files or directories that are not readily cached in a storage system's server memory.

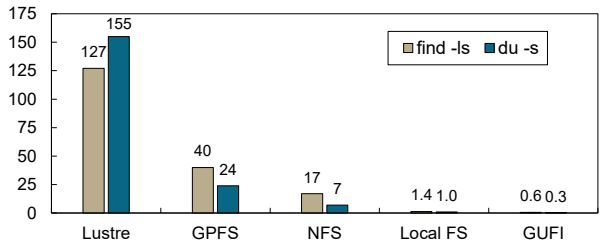


Fig. 1: Time to query the contents of a Linux kernel source tree using different file system technologies. Parallel file systems are inefficient for the small, read-only requests used to query metadata. **GUFI creates an external index that provides local file system like query experience for distributed file system metadata while continuing to exercise standard file system permission control**. Queries by unprivileged users continue to be restricted to data for which they have access permissions while running orders of magnitude faster thanks to a dedicated, query-optimized index.

Figure 1 shows the performance of common metadata queries for file systems often available within large HPC clusters, a local file system, and the GUFI service described in this paper. Each measurement traverses the 74K files in the Linux 5.8.9 kernel source tree and shows the time required to perform recursive directory listings (*find -ls*) and disk usage (*du -s*) for each system's metadata layer. These utilities specifically exercise the `readdir` and `stat` system calls that are the building blocks for all file system metadata queries. Although the file systems all used fast SSDs for metadata storage and had server-side metadata caching; the parallel file

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE

systems, Lustre [11] and GPFS [12], were unable to provide fast metadata query performance. In this paper we describe the efficient search algorithms used by GUFI, a metadata index that is designed to perform file system metadata queries in parallel and generate the concurrency required to achieve high-levels of read performance from SSDs (see §III-C).

Existing metadata search solutions that provide low-latency, searchable indexes for data centers [13–15] rely on database technologies for high levels of query performance but cannot be configured to enforce the hierarchical permissions used by traditional file systems. As a result, these systems cannot be exposed directly to users without sacrificing POSIX permissions. GUFI instead uses a permissions-based sharding of databases to fully enforce permissions so that users — both privileged and unprivileged — can securely access indexed file system metadata. With a cluster-wide user identity control such as LDAP, GUFI easily allows both regular users and cluster administrators to perform queries without requiring additional security schemes to prevent access to private file system and file data metadata (see §III-A).

Finally, existing file systems and file system indexes [14, 16] often lack rich schemas that allow for fast, complex queries across file system metadata and file extended attributes. Due to the hierarchical nature of file system namespaces many common queries emphasize summarizing the contents of file system subtrees or finding the largest files for each user (a set of queries across sibling home directories) – modern metadata systems may accelerate quota enforcement systems but provide few materialized views and indexes to accelerate user queries. GUFI provides a schema tuned to accelerate hierarchical, recursive queries for both users and administrators. Further, the GUFI schema can be extended while in use to accelerate emergent queries (see §III-B).

GUFI is a single unified index for searching metadata and attributes across multiple file systems. The index is constructed by scanning the file systems within a data center and creating clustered embedded SQL databases that are sharded to enforce file system permissions. Because GUFI strictly enforces POSIX permissions it can be directly accessed by users and administrators. The index GUFI builds accelerates interactive command line queries and queries generated by a web-based metadata search across the entire data center. Because the underlying index is stored in a set of embedded databases that are accessed with SQL, GUFI supports complex queries that perform poorly with traditional metadata query tools. Further, because the shards are SQL based, extending GUFI by adding tables, joins, and other familiar mechanisms is simple to accomplish. A unified index sharded by file system access permissions also enables GUFI to easily parallelize query processing across all available shards to achieve high levels of query performance atop fast SSDs.

In addition to accelerating queries by up to 230x compared to the state-of-the-art the GUFI metadata index provides additional advantages over existing scalable file systems and metadata indexes. First, the GUFI system is designed as a data center-wide service capable of indexing and querying the contents of all file systems within a data center. Second, a GUFI index is both composable and decomposable such that any directory or sub-tree of directories within the index can be trivially added, updated, or removed as desired by administrators. Third, the GUFI index provides index construction tools that leverage custom capabilities of common storage systems to support faster index creation and updates. Finally, GUFI leverages open source file systems and embedded databases within its implementation to reduce deployment complexity.

In summary, GUFI makes the following contributions to improving file system metadata queries:

- An index for enabling the rapid search of file system metadata that enforces POSIX permissions to provide secure user and administrator queries,
- A novel index addition that allows extended attributes, which require additional permissions enforcement, to be accessed within the same queries,
- A novel database sharding approach that uses file system ownership and permissions to improve query performance, and
- A performance analysis that demonstrates GUFI query performance in comparison to existing state of the art indexes.

Paper Organization: Section II describes HPC environments where GUFI is deployed with a description of the magnitude of indexed file systems. Section III describes the technical features of GUFI that enable fast queries and secure access for non-privileged users. Section IV evaluates the effective concurrency of GUFI, compares performance with the current state of the art, and describes GUFI's performance for complex queries. Section V describes related work. Section VI summarizes the GUFI system principles.

II. Motivation

The Oak Ridge Leadership Computing Facility and Lawrence Livermore National Laboratory compute facility have described their namespaces as including a billion files and expect those namespaces to service tens of billions of files in the near future[5, 6]. Other large-scale scientific computing sites also produce and store petabytes of data on diverse file systems with namespaces up to billions of files [17–19]. The management of extreme file counts is not unique to scientific computing users with the entertainment industry storing and processing billions of files [20]. These massive file counts and the Petabytes of fast storage associated with the file systems used to match the performance of high-performance computing platforms has caused administrators to rely extensively on quotas and automated purging of older data to maintain storage performance over time [21]. Thus a fast metadata query system is necessary so that facility users can locate and manage data to advance their scientific workflows and act as responsible users of shared storage resources ensuring that the most valuable data is properly archived and that massive temporary data sets do not swell to a size that exceeds quotas (causing batch jobs to fail) or runs afoul of facility capacity and purging policies.

One positive aspect of the strict policy enforcement mechanisms within HPC data centers is that a mature ecosystem of batch tools has emerged for performing full file system scans. System administrators run nightly jobs traversing billions of files to calculate usage statistics and locate stale file system data slated for removal. Further, many local and distributed file systems provide administrator-only utilities supporting fast namespace scans to support these jobs. For example, IBM's Information Lifecycle Management libraries provide a fast inode scan for Spectrum Scale [22] and similar capabilities exist for other distributed storage systems [23–25]. Building schema and indexes that effectively leverage the output of these fast, administrator-only scanning technologies and creating user-accessible query tools is a critical component in enabling agile, policy-compliant data management systems for scientific data.

One critical question is: why can't users simply perform queries directly using the scalable metadata plane provided by distributed file systems? As described in Section I, distributed metadata services are typically tuned to provide the highest performance for creating or opening the newest files from thousands or millions of processes simultaneously. More broadly, fast parallel file systems are designed to provide scale-out performance. While scalable performance is an asset when a scientific application is simultaneously creating and opening a file from 1 million cores, it is of no practical use when a user is trying to search a several million file subtree for a single name from an interactive shell. Parallel search tools that use thousands of compute cores to search for existing files [26, 27] are impractical due to the inefficient use of compute resources. First, these tools consume valuable compute platform time performing basic file system operations rather than using the capability of modern processors to advance scientific simulations and AI-based discovery. Second, these parallel jobs must be scheduled through a batch-oriented resource manager (e.g. the Slurm Workload Manager) and may require hours waiting in a queue before job execution. Modern parallel and distributed file systems are not designed to support the type of efficient, interactive queries required for scientific data management and thus additional tooling is required to provide powerful metadata query capability for HPC data centers and facilities.

The obvious solution for building a queryable metadata resource is to leverage database technology. However, one important difference between databases and file systems is the granularity of access control. File systems support a fine-grained approach to permissions that enables users to create files and directories and also control which other users and groups of users are allowed to access those files and directories. Critically, when a directory is marked as private by a user no other regular user can determine any information about the contents of that directory. This includes being unable to access the names of files within that directory or even knowing if the directory contains files. This type of access control enforcement is difficult to accomplish with database technologies and requires row-level security typically available in enterprise-class Relational Database Management Systems. State of the art row-level security mechanisms result in approximately a 30% performance loss for queries and a 20% performance loss for database updates [28]. An ideal file system metadata querying tool imposes no security-related slowdowns for enabling user-level queries and should instead improve performance by searching only over file system contents a user is able to access rather than performing a scan across all file system entries.

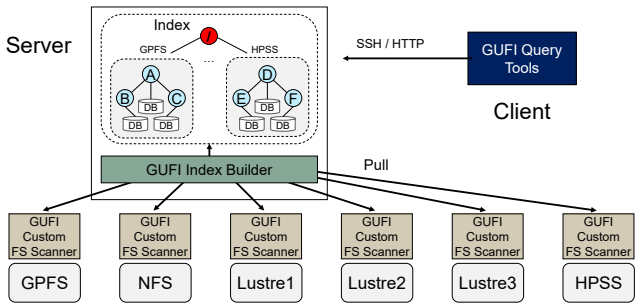


Fig. 2: GUFI system overview. A dedicated server periodically pulls metadata from a set of source file systems, builds indexes as a hierarchy of DB files, and securely processes queries from clients.

III. GUFI Design and Implementation

File system metadata indexes provide two operational benefits to data centers. First, a fast, searchable index makes locating data and orchestrating the movement of data between file systems easier. Second, it moves metadata-intensive queries away from highly-optimized mission-critical file systems and instead directs those queries to a dedicated interactive metadata resource which can significantly reduce performance loss for large parallel jobs [29]. The GUFI index is created to exceed those basic benefits by a) being specifically designed to encode permission information within a file system metadata and extended file attributes index such that the resulting index can be directly and efficiently queried by users (Section III-A), b) providing a rich schema that enables fast data retrieval for the types of queries common across file system subtrees (Section III-B), and c) using a set of algorithms that both reduce the amount of data accessed per query and effectively achieve high levels of performance that match the speed of today's fast SSDs (Section III-C).

A. GUFI Architecture Overview

As shown in Figure 2, the GUFI indexing system consists of a server, interactive query clients, and a set of generic or custom file system scanners (or tree walkers). The server stores all indexes, which are a collection of embedded database files stored in a hierarchy of file system directories on one or more local SSDs. These embedded databases are periodically rebuilt by an GUFI index builder that pulls metadata from a given set of source file systems using either a generic or a custom file system scanner that is optimized for that file system. The index builder runs on the server. Scanners in most cases run as a privileged process on the metadata server of a given source file system. Client searches are performed by tools sending queries to the GUFI server. The server processes the queries. To

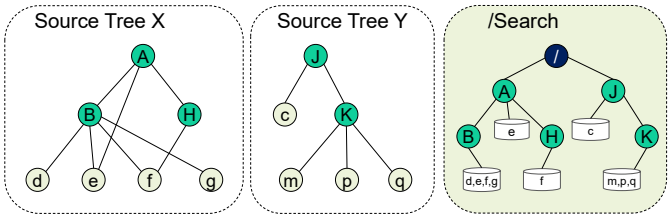


Fig. 3: Index creation overview. Each source tree is recreated, with the same directory structure, permissions, and owners. Directory, file, and symlink metadata are placed into embedded databases and multiple indexes can be created under the same path (e.g. `/Search`), allowing multiple file systems to be searched simultaneously.

fully utilize available storage bandwidth, each query is divided into a large number of sub-queries concurrently executed on the hierarchy of databases stored on the server. By storing the databases in a hierarchy identical to the original source file systems, GUFI is able to enforce the same hierarchical permission scheme used by those file systems and by using databases supporting standard SQL queries GUFI can provide rich capabilities for querying file system metadata.

1) Secure Indexes for System-defined Metadata: POSIX-compliant file systems provide a hierarchical view of the information stored within the file system as directories, sub-directories, and non-directory entries. One commonly misunderstood aspect of POSIX file permissions is how access to metadata entries is controlled. Permission to stat a file or directory (or list its extended attribute *name*) only requires that every parent directory of that entry must be searchable by the user. There is no requirement that the entry itself be readable (though access to extended attribute *values* does require the entry's read bit to be set). Implementing the hierarchy of search bit checks in user space is difficult to accomplish efficiently and requires a careful reproduction of the kernel-enforced file access controls by a privileged process.

As shown in Figure 3, GUFI reuses the operating system file system permissions enforcement and recreates the directory structure of the source file system within the index. File, directory, and symbolic link metadata, such as uid, size, and access permissions, are stored within a single embedded database file stored in each directory of the index. The embedded database functionality is provided by the open-source SQLite library [30]. Notably, the entire GUFI index is simply a collection of database files and directories and it can be managed just like a collection of files and directories. System tools that support snapshots, archives, file transfer tools, and even version control all work seamlessly with the GUFI index. Because GUFI is an index of only metadata, multi-billion file storage systems have resulted in indexes on the order of a few hundred of Gigabytes.

2) Secure Indexes for User-defined Metadata: Extended attributes (XAttrs) are a POSIX feature that enables a name-value pair to be added to files and directories. Extended attributes are increasing in importance within HPC as data centers begin to support increased data labeling efforts for AI workloads. Although XAttr *names* are protected identically to system-defined metadata, XAttr *values* must be protected like file data. Storing XAttr data securely within the index is complex and relies on the following rules:

- A directory's XAttr values are stored within the primary embedded database stored within that directory.
- Any file with identical permissions to its parent directory has its XAttr values stored within the primary embedded database.
- For files with ownership that does not match the parent directory a *per-user* embedded database is created (owned by the respective UID with the GID set to none) to store all XAttr values accessible by that user.
- For files with a group that does not match the parent directory two *per-group* embedded databases are created (owned by the none UID with the GID set to the respective group). One database is used to store group readable XAttr values and another is used to store XAttr values with group read permissions removed.

By setting the ownership and group on the per-user and per-group embedded databases users can only read XAttr values they have access to. We allow users to access all XAttrs stored with their own UID, including those that they do not currently have an enabled read permission. File owners can trivially change their file's permissions on the source file system and thus preventing access to those XAttrs does not increase true security posture. However, as described above, we create two per-group databases to separate XAttr values that are accessible and inaccessible because of group read permissions. In Section III-B we describe how the query tool uses a single database view to aggregate metadata and unique XAttr name-values for each file accessible by a user.

3) Index Creation Tools: In order to construct input data for GUFI it is necessary to perform a metadata scan of each source file system included within the index. This scan is performed as a privileged user to ensure that permissions do not prevent access to any parts of the source file system.

In the worst case, the time to create an input data set for an index requires a full tree walk of the source file system which is performed within the source file system metadata servers to minimize access latency. Naive scan tools may take several hours to scan billions of files and directories. Fortunately, many local and distributed file systems provide administrator-only utilities that provide faster namespace scans. The GUFI toolset includes scanners that leverage those interfaces if they are available. For example, GUFI leverages IBM's Information Lifecycle Management libraries to provide a fast inode scan for Spectrum Scale [22]. Similar capabilities exist for other storage systems [23–25]. For network file system implementations that provide snapshot capabilities, such as WAFL [31] and ZFS [32], the GUFI scan tools leverage a consistent snapshot to produce an accurate index of the metadata state. However, even with custom scan tools that can perform scans in minutes rather than hours, the GUFI index must be at least several minutes old if a scan of the active metadata system is performed. Specifically, large data movements that are in process during source file system scans will not be well characterized.

TABLE I: Examples of file system scanning and index creation times. All scans were performed using a single node (either via client mount or running on the metadata server directly). The measured times are not directly comparable because the concurrent load and hardware platforms are not equal. In the most basic case GUFI relies on a threaded breadth-first tree walk to scan source file systems. Only one file system (Scratch1) was configured to use the lazysize mount option and relied on an underlying format compatible with lester [33]. When tree walks must be used to scan larger file counts the scanning processes can be distributed across multiple clients. Index creation either occurs in-situ (concurrent with the scan) or as a post processing step after the scan.

Filesystem	Type	Dirs	Files	Scan Type	Sampled Scan Time	Index Creation Time
/users	NFS	6.1M	43M	Tree Walk	50m	In-Situ
/proj	NFS	35.7M	263M	Tree Walk	133m	In-Situ
/scratch1	Lustre	7.4M	102M	Lester	19m	158s
/scratch2	Lustre	16.5M	225M	Tree Walk	216m	In-Situ
/archive	HPSS	5.7M	193M	SQL	125m	229s

In order to update stale index data GUFI includes a tool that enables a user to request an immediate update to the index and directory permissions for a single directory. This tool was designed for file transfer utilities used to migrate data between file systems. However, it has since been used to support cases where a user realizes they have exposed sensitive information within metadata and must immediately enforce a change of visibility or metadata contents. In this case the tool is used to trigger an immediate index update and preserve user-enforced security for metadata visibility.

4) Online Index Updates: The GUFI index updates are based on a pull process that retrieves the most up to date source file system scans at a configurable interval. Based on the scan times shown in Table I our facility uses an index update interval of 4 hours. This update frequency was based on the practical rate at which the slowest source file systems could be scanned without requiring a large number of client nodes to perform scans. Large numbers of nodes participating in the scan are more likely to result in large metadata loads that affect batch jobs running within the data center. Some source metadata systems, for example large tape archives where the SQL-based scanning technology cannot be parallelized, may not be able to produce a full trace within the interval window. In this case GUFI pulls and applies the most up-to-date completed scan data to the index. Fortunately, in many cases index creation can be effectively overlapped with the scan and thus the pull process is able to retrieve a fully created GUFI index. Even in the case where the index is constructed as a post processing step at the GUFI server the creation time is inexpensive because we rely on the performance of local file systems to create and store the index. As the last column in Table I shows our commodity servers using in-kernel file systems (e.g. XFS [34]) are able to create 1M directories with databases in approximately 18 seconds and insert 100M index rows in less than 120 seconds. The update process is a rename of a symbolic link that points at the new index to overwrite the symbolic link pointing at the stale index allowing existing queries and new queries to make progress simultaneously. While bulk scanning drawbacks are significant, having two complete namespace snapshots available that are separated by a few hours also enables new query types that can passively measure data movement within and between file systems. Further, within many cloud and HPC data centers the largest driver of namespace and data mutations are parallel batch jobs executed non-interactively by a scheduler and users are aware that even on live file systems namespace queries include out-of-date data.

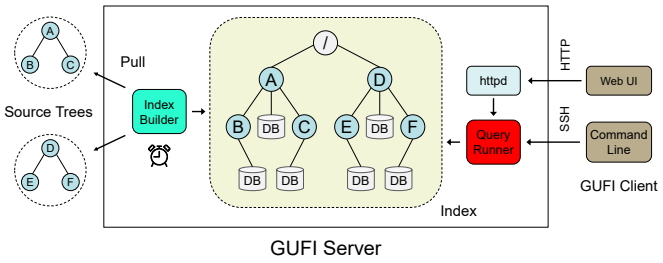


Fig. 4: GUFI deployment. Indexes are built and updated at regular intervals. Users perform queries using a web interface or scripts that SSH into the server. Users are not expected to log into the server through interactive terminals.

5) Index Access Controls: Because GUFI implements a single shared index for the entire data center – including for administrative use – it is critical that the index is not corrupted by users. Thus access to the index must be read-only for users even though the index directory permissions includes read and write permission settings (preserving the original directory permissions is critical to generating correct query results). One solution to providing read-only access to the index is to perform a read-only mount of the index directly onto client systems. However, during deployment a read-only mount led to user confusion as duplicated directory trees are available with source directories containing files and identical index directories that contain embedded databases. We also developed a FUSE interface for GUFI but were unable to achieve satisfactory query performance.

Figure 4 shows how the index is currently deployed within the data center to enable remote, read-only access while supporting access via the parallel toolset. Each client node that has access to the index creates an empty directory (typically named /search) that can be used as input to the GUFI command line tools. When a user directs a `gufi_find` or `gufi_ls` at the /search directory, the index is consulted by sending a remote invocation from the client node to a server that hosts the index. This access is secured using a restricted shell that allows only GUFI tools to be invoked at the server. Web access interacts similarly, with the user's web browser requesting authenticated access to a search prompt and sets of pre-generated queries are available that describe various metadata characteristics such as the user's largest files and their most recently accessed files. Every query requires full authorization with the authentication service (e.g. LDAP) so that changes to the user's access rights are registered immediately and all queries obey the most up-to-date system access privileges.

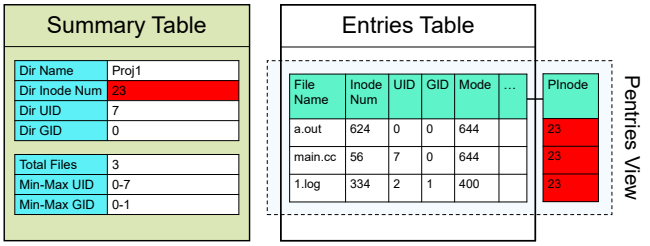


Fig. 5: Index schema for GUFI. The pentries view provides parent inode as a column alongside the entries table.

Finally, to prevent regular users from performing queries that modify the live GUFI schema all of the query tools that are user facing can only open the embedded database files with the read-only flag (`O_RDONLY`). While the ability to modify the database schema is a powerful capability, we only allow administrator query tools to open database files with the write flags required to enable that feature.

B. Index Schema

The database schema was constructed to efficiently answer many types of queries and provides three types of SQL record holding tables:

- `entries`: A table to store the metadata about about each directory entry (files and links).
- `summary`: A table that describes current directory as well as the characteristics of the current directory's entries.
- `t`[ree]`summary`: A table that summarizes the characteristics of the entire tree starting at the current directory.

While the `entries` table is a reproduction of the metadata attributes commonly associated with directory entry inodes, the two types of summary tables merit further description. The `summary` table describes information about the current directory including the minimum file size, maximum file size, number of files, and total directory size in addition to the directory's own metadata. The additional data provided by the `summary` table significantly reduces the costs of common queries seeking to determine how much space a sub-tree uses or to locate the smallest/largest file within a sub-tree. `tsummary` tables are not created by default as the index is constructed. Instead the administrator must trigger the process to build each one. The resulting table summarizes the sizes, user counts, group counts, and additional information for every file and directory beneath the table. Both `summary` and `tsummary` tables can have overall, per-user, and per-group records thus making per-user or per-group summary queries extremely efficient as well.

GUFI also provides persistent views of tables with altered schemas to make query construction simpler. One such view is the `pentries` view which, as shown in Figure 5, is the `entries` table augmented with the parent inode column.

Because the index is stored within embedded SQL databases there are few practical limits to what fields can be added to the index. The same tools that query the index can be used to add tables and views or alter the schema. While we don't expect many administrators to take advantage of this capability, it is straightforward to copy an index to a scratch location, recursively modify the schema to support a custom query, and include that change as part of a process similar to the `tsummary` construction.

1) Extended Attribute Schema: XAttr names are stored as a list within a column in the `entries` table. The XAttr table schema is a table containing 2 columns: the entry's inode that the XAttr is associated with and an XAttr value list. Tables with this schema are not queried directly. Rather, a view of all unique accessible XAttrs is created by performing a union on the directory database's XAttr table with all successfully opened per-user and per-group XAttr database files within the current directory. Per-user and per-group XAttr databases that the user is unable to open are not accessible and thus not included within the dynamically constructed database view. Because different users will generate different views, we do not provide persistent views of the XAttrs. For convenience, we automatically generate temporary views combining `entries`, `pentries`, and `summary` with the XAttrs view.

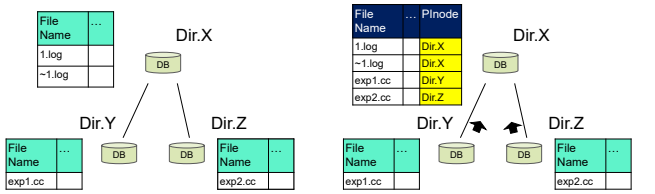
As an optimization, an additional table is created in the directory's database during XAttr indexing that keeps track of the per-user and per-group XAttr database files that were generated. This removes the need to traverse the directory in search of XAttr database files to attach.

C. Parallel Index Algorithms

GUFI provides parallel versions of tools that reproduce the behavior of common file system utilities, a web interface that provides a search bar and a set of utilities for building and managing the index. These tools are all built on a parallel tree descent code base that is designed to open directories and files rapidly while performing data aggregation at multiple points during request processing. This code base descends file system trees in breadth-first order with each directory being processed by a single thread and sub-directories added to a queue for subsequent processing by the thread pool.

1) Source File System Scans: The source tree scan processes the directories of the source tree in breadth-first order in order to build a list of all databases to create in parallel. As each directory is added to the work queue a single thread is assigned to process the contents of each directory. To convert these processed trace files into directories and databases, GUFI provides a parallel ingest tool that creates directories as-needed while inserting entries into the embedded databases. GUFI also provides a parallel ingest tool that skips the list building stage and generates databases as source directories are encountered.

2) Parallel Index Queries: The `gufi_query` executable directly accesses the GUFI index and is used to build tools such as `gufi_find` and `gufi_ls`. Because the permissions of the directories are preserved within the index structure, the query tools only search (in breadth-first order) databases which the user can access. Thus, user query performance is proportional to the amount of metadata that is accessible to the user rather than the amount of data that is stored within



(a) Without rolling up, traversal of this index requires 3 threads, 3 dir opens, and 3 DB opens.

(b) With rolling up, traversal of this index requires only a single thread, a single dir open, and a single DB open.

Fig. 6: Comparison of traversing an unmodified index and an index that has been rolled up.

the entire index. Further, since the index schema includes summary information many queries do not need to access more than the pre-computed summary tables.

Query processing begins at the user provided directory, the sub-directories of the provided directory are discovered and pushed into the thread pool for processing. Then, SQL queries provided to `gufi_query` are executed on the discovered directory databases. Complex SQL queries requiring advanced operations such as aggregation, uniqueness checks, and reduction operations are supported. Further processing may be done within `gufi_query` to aggregate results between queries instead of printing results from each directory's database as they are discovered. The per-directory results are written to per-thread in-memory databases to avoid contention resulting from multiple threads inserting into a single database. After index traversal has completed, a user provided SQL query is executed to merge the per-thread result(s) into a single set of results. A final user provided SQL query is then run on the merged results to print out the final result(s).

3) Permissions-based Sharding: By default, a GUFI index replicates the shape, ownership, and permissions of the source tree directories. While this scheme makes enforcing metadata access controls simple, the database per directory scheme often results in millions of small databases which are inefficient to query. To improve performance, GUFI identifies opportunities where an entire sub-tree of directories have compatible access rights and merges the data from the sub-directories into the database in the parent directory. When the access rights of a sub-tree are compatible with access to the database at the top of the sub-tree the sub-tree entries can be merged into the top-level database which then summarizes a larger amount of metadata. This optimization both reduces the number of databases accessed during queries and increases query request sizes when compared to reading many smaller databases. We call this merge operation a GUFI index *rollup*.

In order for a target directory rollup to be valid, two conditions must be met. First, all of the sub-directories below the target directory must be rolled up. [1] Second, all target directory and sub-directory pairs must have permissions that satisfy any one of the following conditions:

1) World readable and executable (i.e. o+rx is set)
2) Matching permissions (user, group, and others), with the same user and group
3) Matching user and group permissions, readable and executable (ug+rx) with the same user and group, and not world readable and executable (i.e. o-rx is set)
4) Matching user permissions, readable and executable (u+rx) with the same user, and not group or world readable and executable (go-rx)

If a directory does not fulfill the above conditions, its `summary` table is marked to indicate that it is not rolled up. If a directory can be rolled up, the directory's database is modified. The `pentries` view is dropped, and replaced with the `pentries` table with identical columns and contents as the original view. The `pentries` table of each sub-directory is then copied in. This allows for the rows of the sub-directories to be inserted into `pentries` without corrupting the original data located in the `entries` table. Each sub-directory's `summary` table is copied into the directory's `summary` table, with the directory's name prefixed to the sub-directory's name and is marked as not being the original row in the table. The directory is then marked as having been rolled up. This series of operations is performed recursively up the tree. Rollup of per-user and per-group XAttr databases is performed similarly using the same permission checks used for rollup XAttr databases.

The databases and directories that are rolled up are not removed. This has two useful effects. First, because all of the original directories exist and contain rollup data, queries can start at any directory and take advantage of the rollups that exist within that subtree. Second, if a rollup needs to be undone (for example, to perform a user-requested directory update for an index that has already been rolled up), the process to do so is more lightweight than recreating the entire subtree. In addition to not recreating each directory of the subtree, each directory's rollup can be independently undone due to not having any dependency on any other directory's rollup data. The steps for removing a directory's rollup are as follows: drop the `pentries` table, restore the `pentries` view, and delete the rows in the `summary` table that were not originally there.

IV. Evaluation

We have implemented GUFI using 14K lines of C code. We perform four sets of experiment to evaluate GUFI. We first show that GUFI is able to transform high-level data management queries into concurrent, low-level disk reads that are parallel enough to fully saturate one or even more underlying SSDs, effectively converting a traditionally non-disk-bottlenecked workload to be disk-limited. Metadata queries are traditionally not limited by disks due to remote processing (metadata is accessed through clients sending RPCs to servers), concurrency control (which tends to serialize concurrent metadata requests for atomic execution), and the use of on-disk layouts that are primarily optimized for per-file inode accesses. As a result, today's metadata storage servers are often unable to keep disk command queues full, causing

[1] Leaf directories are, by definition, considered rolled up.

TABLE II: Evaluation configurations. The hardware, software, and data set used to perform the GUFI evaluations. The data sets were extracted from real file systems within an HPC data center.

Server	HPE DL380
Processor	Dual Intel Xeon 8280 2.7GHz (56 total cores, 112 threads)
Main Memory	192GiB
Storage	4x Samsung 1725A 1.6TB
Operating System	CentOS 7.7.1908
Kernel Version	Linux 5.7.9
File System	XFS
Dataset 1	1.6M directories, 13.2M files
Dataset 2	2.2M directories, 64.7M files

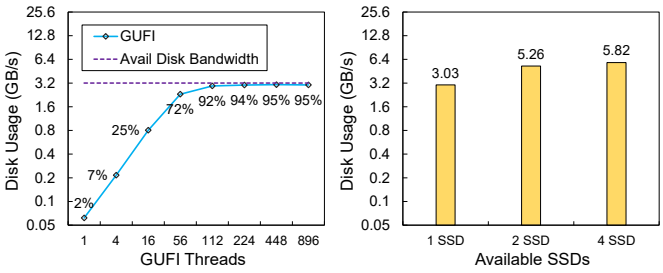
available disk bandwidth resources to rarely be fully utilized to enable rapid query performance. By establishing a separate, sufficiently sharded index using a set of compact SQLite databases clustered on a single, dedicated storage server and by allowing both regular and administrative users to directly log onto that server for query execution, we show that GUFI is capable of generating a much more parallel workload that more effectively exposes underlying SSD capabilities.

Next, we focus on GUFI's permission-based rollup mechanism and perform sensitivity tests to measure its effectiveness in reducing final index size and preventing an excessive amount of small SQLite databases which tend only to reduce query performance.

Our third set of experiments involve micro benchmark tests that use extended file attribute queries as a proxy to measure and compare GUFI's query performance with XFS, the current state-of-the-art for local file system metadata management.

Finally, we perform macro benchmark tests on GUFI using a real production file system namespace and compare its performance with Brindexer [15], one of the current state-of-the-art technologies for file system metadata indexing. Brindexer relies on many of the same technologies as GUFI, including SQLite and parallel query tools that rely on a thread per database to execute queries with a parent process merging the results of each thread. Unlike GUFI, Brindexer uses a hash-based data partitioning scheme that uses the parent directory of each file to hash the metadata entry to one of 256 embedded databases. Brindexer is reported to be the highest performing file system metadata index. It currently cannot enforce standard user-oriented permission access control.

Table II shows the hardware configurations, system software versions, and the data set characteristics used for the evaluation experiments. Dataset 1 was generated by running GUFI scan on an NFS file system and anonymizing the output. Dataset 2 was generated by GUFI scan on a Lustre file system within an HPC data center and includes all of its real metadata. Our evaluation focuses on GUFI's query performance. For each experiment run, the GUFI index is stored on a local XFS file system [34]. A modified SQLite [30] 3.27.2 (enabling optimized read-only access) was used to implement GUFI's embedded databases.



(a) Single-SSD Performance (b) Multi-SSD Performance

Fig. 7: GUFI transforms file system metadata queries into massively parallel bulk disk reads fully exposing the potential of one or even more underlying SSDs for rapid query performance until the host server becomes a performance bottleneck.

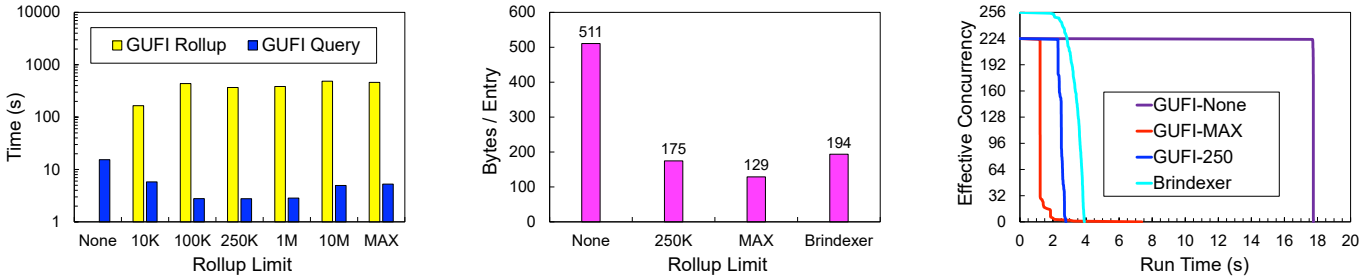
A. *Underlying Disk Utilization*

To attain high query performance, GUFI creates one database per directory and uses a pool of threads to process each database in parallel resulting in high levels of concurrent disk read operations. To demonstrate GUFI's effectiveness in fully utilizing an underlying SSD, we use dataset 1, an GUFI index constructed from a production home file system that has been anonymized. Because the anonymized file names and users are no longer representative of the original namespace we only use this dataset to evaluate hardware utilization. We vary the GUFI thread pool size from 1 to 896 (numa-aligned with the host processor which had 28 CPU cores). To collect the disk usage achieved by GUFI we first drop all caches and then use a block tracing program to capture disk accesses during each query. Figure 7a shows that in the case where the index is stored on a single SSD, sufficient concurrency can be achieved using 112 threads to effectively saturate the disk.

In Figure 7b we increase the available storage performance ceilings to 6.4 and 12.8 GB/s by using two and four SSDs respectively. With two SSDs we again see that GUFI generates a high level of disk read performance resulting in a disk throughput of 5.26 GB/s (82% utilization). On the other hand, with four SSDs we supply substantial surplus disk performance that the index is unable to use due to the host server becoming a performance bottleneck. In order to use four SSDs effectively GUFI may be configured with two index servers each with two SSDs. However, multi-host configurations are beyond the scope of this evaluation section and results presented in the remainder of this section use the GUFI server configured with two SSDs and 224 threads.

B. *Permissions-based Database Rollups*

Although a single GUFI index is capable of returning metadata at rates approaching disk speed limitations for a single host, it is still important to leverage the rollup process within GUFI. This is because rollups address four sources of index processing overhead: time spent opening databases, time spent reading database metadata, time spent setting up queries, and small average disk read sizes. For our system configuration (which uses fast processors, fast SSDs, and fast local file systems) open performance, request sizes, and query setup



(a) Rollup and query times

(b) Space overhead

(c) Concurrency over time.

Fig. 8: Rollup limit tradeoffs. In (a) we show the performance of the rollup process at multiple rollup limits and the performance of a simple query with the resulting index, in (b) we see how well the database space overhead is amortized at 3 GUFI rollup limits and for the Brindexer index, and in (c) we show thread completion times and how a small number of large databases lead to lower effective concurrency.

times are not significant bottlenecks; however, even an empty SQLite database includes 12KB of data that must be read. Without any rollup process a source file system namespace containing 1 million directories will result in 12GB of database metadata that must be read and processed along with GUFI queries. Compared with existing indexes such as Brindexer that hash data across typically only a few hundred databases per file system, GUFI— without any rollups — may result in 100× as much data retrieved for every query.

To evaluate the effectiveness of rollups we ran them against 5 production file system namespaces ranging from home, project, scratch, and archival storage spaces (see file system counts in Table I). Our measurements showed that rollup achieves an average of 386x reduction in the number of databases compared with an GUFI index without an rollup. The largest reduction was observed at a home space (741x) while the lowest rollup rate was seen at a project storage space (77x) due to a more diverse spread of user-group-permission combinations.

While being vital in reducing the cost of processing each database, rollups can also reduce performance if insufficient concurrency is available after the rollup. Figure 8 creates indexes using the dataset 2 production scratch file system to show the tradeoffs that exist within the GUFI rollup process. The NONE index has not been rolled up while the MAX rollup has had no limit placed on its rollup process. Figure 8a shows the time required to perform the rollup process at a variety of rollup limits and the effect of that rollup limit on execution time for a simple query. We dropped all caches before each measurement and report the average time across five runs. At rollup limits greater than 10K entries the rollup process requires between 367 to 485 seconds. In particular, we note that a rollup limit of 250K entries results in the lowest rollup time (367 seconds) and the lowest query time (2.6 seconds).

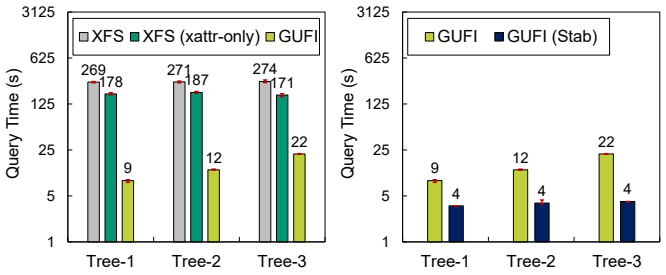
Figure 8b shows the per-entry index cost across different rollup configurations. For reference we also include the space overhead of Brindexer which hashes index entries across 256 databases. As expected, we see that as the rollup limit is increased the total number of databases is reduced and the Bytes/entry is similarly reduced. The reason GUFI 250K and GUFI MAX — which have larger database counts than Brindexer — have lower total database sizes is that for Brindexer it is necessary to store full parent directories for each entry while GUFI does not require this overhead: GUFI's sharded databases preserve the structure (and the permission information) of the source directory tree.

Figure 8c shows the effective concurrency when requesting 224 threads for use by GUFI to complete a query that touches all files. The line shows the time when each thread completes the last database it is able to process with GUFI. We see that although GUFI NONE is able to keep effective concurrency high, the large amount of index data that it has to open and read result in nearly 18 seconds to perform the query. On the opposite end of the spectrum we see that GUFI MAX has the lowest amount of data to process, but the large amount of time required to process a single large database dominates the 8 second execution time.

Both GUFI 250K with 224 threads and Brindexer with 256 threads (i.e. thread per database) are able to achieve a better balance of space overhead combined with high levels of effective concurrency. In addition to having lower database overhead GUFI 250K also exhibits better effective concurrency. Because the Brindexer scheme uses a hash of the parent directory to distribute data to individual databases and large directories are true outliers, being both very large and very uncommon, the Brindexer database sizes range from 24MB to 80MB. GUFI 250K results in 34K databases where the largest database after rollup is only 29MB leading to greater effective concurrency and lower overall execution time. For all remaining performance tests we use the GUFI index with rollup databases limited to 250K entries.

C. Extended Attribute Query Performance

Next, we evaluate the performance of GUFI's extended attribute query capability. To establish baseline performance we use a standard XFS file system on the same server as the GUFI index with the file system stored on a single SSD (query performance for XFS was not improved by striping across two SSDs). We began with dataset 2, an unmodified index of a scratch filesystem with 2.2 million directories and 64.7 million files and we have added extended attributes to the index. Tree-1 has 25% of the total files populated with XAttrs, Tree-2 has 50% of the total files populated with XAttrs, and Tree-3 has XAttrs specified for 100% of the files. While the



(a) XFS vs GUFI on Scans (b) Scan vs Stab Performance

Fig. 9: Extended attribute query performance. In (a) we show the time to search for an XAttr present in 25% of the files (Tree-1), 50% of the files (Tree-2), and 100% of the files (Tree-3). Compared to searching for XAttrs using XFS both with and without a recursive tree walk to locate files the GUFI index improves performance by up to 33x. In (b) we compare the performance of querying for an XAttr that exists in each file with XAttrs (scan) and a unique XAttr that exists in only a single file (stab) and find up to a 5.5x speedup for highly selective stab queries.

XAttrs data for the files is randomly generated, a well-know sentinel XAttr is always included as one of the XAttr name-value pairs.

Figure 9a shows the average performance across 5 trials performing a full file system search to find all files that have an XAttr name and value matching the sentinel name-value pair (i.e. query selectivity is 25% for Tree-1, 50% for Tree-2, and 100% for Tree-3). For the XFS file system we measured two query approaches: performing a find command paired with a getfattr command per file and using a pre-generated list of all files in the file system to only perform the getfattr command. In both cases performance is proportional to the total number of files rather than the number of files with XAttrs because there is no POSIX system call to select files with XAttrs. Using GUFI to query for files with an XAttr matching the sentinel is significantly faster and results in performance proportional to the number of files with XAttrs. GUFI achieves a 33x speedup for Tree-1, a 22x speedup for Tree-2, and a 12x speedup for Tree-3 compared to the find+getfattr baseline using XFS. Figure 9b compares the performance of using GUFI to search for the sentinel XAttr and locating a unique XAttr present in a single file. Locating and returning only a single XAttr improves GUFI query performance an additional 2-5x dependent upon the number of files with attributes.

D. Macro-Benchmark Results

Finally, we report macro benchmark results that compare GUFI with the current state-of-the-art metadata indexing engine, Brindexer. In order to fairly evaluate the performance of a file system index it is necessary to use realistic data and realistic queries. This is because that the size of file and directory names as well as the depths and sizes of directories all directly effect the design and performance of a file system index: an index that is only evaluated using compact file names will not truthfully reflect the overheads associated with storing the full path names of all indexed entries.

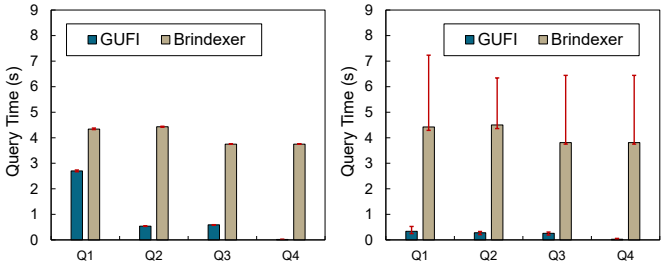
To compare the performance of GUFI and Brindexer we use dataset 2, a real namespace collected from a production scratch file system with 2.2 million directories and more than 64 million files. We configure Brindexer to use 256 threads and 256 databases organized such that each thread is responsible for processing an entire database. The GUFI index has been rollup processed with a limit of 250K entries as the largest possible rollup table size and 224 threads. To evaluate the performance of GUFI and Brindexer we use the following queries:

1) List all file names accessible by the user.
2) Print the size and name of every directory accessible by the user.
3) Print the space used by the users home directory (top-level for root).
4) Print the space used by the users home directory (top-level for root).

These queries are commonly executed using standard utilities such as ls, find, and du for file systems supporting those tools. The last two queries are duplicates however we use different processes to execute the queries within GUFI. In the query 3 case we execute the command by accessing multiple databases and summing the per-directory `summary` sizes. In query 4 we rely on the `tsummary` table within GUFI's schema to access the size at the top-most level.

In Figure 10a we dropped caches and executed each of the queries using the root user with 5 trials per query reporting the average, minimum, and maximum times observed. This is the fairest method to compare between the existing state of the art because most available indexes cannot be directly used by users. The GUFI index is faster for all queries with speedups on average of 1.5x, 8.2x, 6.3x, and 230x respectively. The final speedup is because GUFI needs to access only a single entry within the `tsummary` table in order to determine the size of the tree. The time required to build the `tsummary` at the top-level directory was 14.8 seconds if done before the rollup process and only 0.368 seconds if done with the 250K rollup index.

In Figure 10b we execute the same queries in the same fashion using 150 randomly selected user IDs reporting the average time across all users as well as the fastest and slowest user query times. This is approximately 10% of the total users that own directories within the system and includes users with both small and large amounts of data. To calculate the Brindexer query times we augment the Brindexer query with a UID in the SQL WHERE clause, but note that the amount of data Brindexer accesses does not change because the query must traverse the entire index. The queries executed on the GUFI index use the substitute user command to first become that user, and then the query is executed which means that only user accessible data is traversed within the index. We see that the Brindexer performance is virtually identical to the admin queries. Although we observed a large outlier in the Brindexer performance, we believe that this was due to an internal SSD task as we were not able to reproduce that outlier in multiple runs. Although the rollup process often results in very few databases for a single user, the benefits of reducing



(a) Admin Query Times (b) User Query Times

Fig. 10: Comparison of GUFI and Brindexer Query Performance. GUFI provides average speedups of 1.5x - 230x for administrator compared to the Brindexer index using a real file system namespace. User-specific queries result in even greater speedups; however, only GUFI can be accessed safely by users.

the search space outweigh the loss of concurrency and single user query performance results in even further performance benefits compared to Brindexer.

V. Related work

File system metadata performance has long been recognized as an important focus within file system performance. Common local file systems use B-trees for storing and processing metadata information [34–36]. TableFS [16] uses an LSM tree to insert metadata into sorted and indexed logs to enhance insertion performance. These metadata systems are developed to address diverse file system workloads but are not designed for metadata queries from a large number of clients. Scalable distributed file system metadata planes including IndexFS [7], DeltaFS [8] and InfiniFS [10] are focused on accelerating file insertion, opens, and deletion rather than supporting scalable metadata queries. Archival storage systems, such as HPSS [25] and Haystack [37], leverage relational databases to create their metadata store. These systems are designed to enable privileged processes to efficiently locate single items from within a store. GUFI is not a general purpose metadata plane but is instead tuned to provide high levels of performance for both small and large metadata queries.

External metadata indexes are a popular technique for accelerating file system metadata performance. The Robinhood Policy Engine [14] uses a relational database to store and query metadata for the Lustre file system. Starfish [13] and BorgFS [38] hash data across multiple databases and are designed to support multiple file systems. HPE DMF7 [39] maintains namespace reflection of file and directory metadata in Cassandra [40] which can be queried using Spark [41, 42]. Spyglass [43] relies on subtree partitioning and snapshots to distribute metadata across a collection of index databases. Brindexer [15] provides better performance than these systems by relying on hash-based data partitioning, a flattened schema, and parallel query tools. These external metadata indexes enable rich metadata queries but require complex joins to check permissions on parent directories and cannot be accessed by users. GUFI provides best-in-class performance while also supporting secure user queries.

Many alternative data systems exist for providing the ability to tag data with additional metadata and provide a search function across that metadata. Enhanced metadata systems like Tagit [44], SoMeta [45], and EMPRESS [46] allow users to define custom tags as indexes for locating data sets of interest. Haystacks does not rely on tags and instead indexes the textual contents of files [47] similar to technologies such as ElasticSearch [48]. These indexes are all designed for open user access and focus on searching within user or content defined metadata while GUFI is focused on strictly enforcing permissions as well.

VI. Conclusion

As data centers continue to improve support for complex workflows and AI-based scientific inquiry, the ability to to search for curated and labelled data sets across the entire data set will increase in importance. GUFI is designed to make searching for data sets within vast data center stores faster, simpler, and more powerful than existing technologies. The GUFI metadata index organizes file system metadata and extended attributes so that all of the file systems within a data center can be securely queried with the highest levels of interactive metadata performance. When measured using real file system namespace data collected from a large-scale data center GUFI provides performance that is hundreds of times faster than existing metadata indexes while using less space and enabling direct access by data center users.

The GUFI file system metadata index relies on three primary enhancements for improving the performance of metadata queries. First, GUFI provides a permissions-aware database partitioning scheme that enables users to only search through data that is accessible by file system permissions. Second, GUFI provides a directory rollup mechanism that limits the space overhead associated with having a database per directory while also enabling sufficient concurrency to saturate multiple SSDs with metadata query workloads. Finally, GUFI provides a rich schema that is designed to accelerate the types of queries that are commonly performed by both users and administrators for single and multiple file systems.

Acknowledgements

We would like to thank the anonymous reviewers for comments improving descriptions and clarity within this manuscript. This manuscript has been approved for unlimited release and has been assigned LA-UR-22-28903. The paper has been authored by an employee or employees of Triad National Security, LLC (Triad) under contract with the U.S. Department of Energy (DOE). Accordingly, the U.S. Government retains an irrevocable, nonexclusive, royalty-free license to publish, translate, reproduce, use, or dispose of the published form of the work and to authorize others to do the same for U.S. Government purposes.

References

[1] *APEX workflows*, https://www.nersc.gov/assets/apex-workflows-v2.pdf, Mar. 2016.

[2] J. Lüttgau, S. Snyder, P. Carns, J. M. Wozniak, J. Kunkel, and T. Ludwig, "Toward understanding i/o behavior in hpc workflows," in *Proceedings of the 2018 International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS 18)*, 2018, pp. 64–75. DOI: 10.1109/PDSW-DISCS.2018.00012.

[3] R. B. Ross, G. Amvrosiadis, P. Carns, C. D. Cranor, M. Dorier, K. Harms, G. Ganger, G. Gibson, S. K. Gutierrez, R. Latham, B. Robey, D. Robinson, B. Settlemyer, G. Shipman, S. Snyder, J. Soumagne, and Q. Zheng, *Mochi: Composing data services for high-performance computing environments*, 2020. DOI: 10.1007/s11390-020-9802-0.

[4] G. K. Lockwood, S. Snyder, S. Byna, P. Carns, and N. J. Wright, "Understanding data motion in the modern hpc data center," in *Proceedings of the 2019 International Parallel Data Systems Workshop (PDSW 19)*, 2019, pp. 74–83. DOI: 10.1109/PDSW49588.2019.00012.

[5] S.-H. Lim, H. Sim, R. Gunasekaran, and S. S. Vazhkudai, "Scientific user behavior and data-sharing trends in a petascale file system," in *Proceedings of the 2017 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 17)*, 2017. DOI: 10.1145/3126908.3126924.

[6] F. Wang, H. Sim, C. Harr, and S. Oral, "Diving into petascale production file systems through large scale profiling and analysis," in *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS 17)*, 2017, 37–42. DOI: 10.1145/3149393.3149399.

[7] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion," in *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 14)*, 2014, pp. 237–248. DOI: 10.1109/SC.2014.25.

[8] Q. Zheng, C. D. Cranor, G. R. Ganger, G. A. Gibson, G. Amvrosiadis, B. W. Settlemyer, and G. A. Grider, "Deltafs: A scalable no-ground-truth filesystem for massively-parallel computing," in *Proceedings of the 2021 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 21)*, 2021. DOI: 10.1145/3458817.3476148.

[9] Y. Wang, C. Li, X. Shao, Y. Chen, F. Yan, and Y. Xu, "Lunule: An agile and judicious metadata load balancer for cephfs," in *Proceedings of the 2021 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 21)*, 2021. DOI: 10.1145/3458817.3476196.

[10] W. Lv, Y. Lu, Y. Zhang, P. Duan, and J. Shu, "InfiniFS: An efficient metadata service for Large-Scale distributed filesystems," in *20th USENIX Conference on File and Storage Technologies (FAST 22)*, Santa Clara, CA: USENIX Association, Feb. 2022, pp. 313–328. [Online]. Available: https://www.usenix.org/conference/fast22/presentation/lv.

[11] P. J. Braam and P. Schwan, "Lustre: The intergalactic file system," in *Ottawa Linux Symposium*, vol. 8, 2002, pp. 3429–3441.

[12] F. B. Schmuck and R. L. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST 02)*, 2002, pp. 231–244.

[13] J. Farmer, "Starfish: A side-band database for hpc and archival storage systems," Santa Clara, CA, 2017. [Online]. Available: https://www.storageconference.us/2017/Presentations/Farmer.pdf.

[14] T. Leibovici, "Taking back control of HPC file systems with robinhood policy engine," *CoRR*, vol. abs/1505.01448, 2015. arXiv: 1505.01448. [Online]. Available: http://arxiv.org/abs/1505.01448.

[15] A. K. Paul, B. Wang, N. Rutman, C. Spitz, and A. R. Butt, "Efficient metadata indexing for hpc storage systems," in *Proceedings of the 2020 IEEE/ACM International Symposium on Cluster, Cloud, and Internet Computing (CCGRID 20)*, 2020, pp. 162–171. DOI: 10.1109/CCGrid49817.2020.00-77.

[16] K. Ren and G. Gibson, "TABLEFS: Enhancing metadata efficiency in the local file system," in *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, 2013, pp. 145–156.

[17] NERSC. "Storage trends and summaries." (2021), [Online]. Available: https://www.nersc.gov/users/job-logs-statistics/storage-and-file-systems/storage-statistics/storage-trends/.

[18] NASA. "Data storage systems." (2021), [Online]. Available: https://www.nas.nasa.gov/hecc/resources/storage_systems.html.

[19] A.-J. Peters. "Eos - the cern disk storage system driving cernbox." (2017), [Online]. Available: https://indico.cern.ch/event/565381/contributions/2401957/attachments/1404011/2144404/CS3-EOS.pdf.

[20] NetApp. "Dragons, devops,multi-cloud: Netapp insight." (2018), [Online]. Available: https://e3zine.com/devops-multi-cloud-netapp/.

[21] W. Zhang, S. Byna, H. Sim, S. Lee, S. Vazhkudai, and Y. Chen, "Exploiting user activeness for data retention in hpc systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21, St. Louis, Missouri: Association for Computing Machinery, 2021. DOI: 10.1145/3458817.3476201. [Online]. Available: https://doi.org/10.1145/3458817.3476201.

[22] M. Beigi, M. Devarakonda, R. Jain, M. Kaplan, D. Pease, J. Rubas, U. Sharma, and A. Verma, "Policy-based information lifecycle management in a large-scale file system," in *Sixth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'05)*, 2005, pp. 139–148. DOI: 10.1109/POLICY.2005.26.

[23] D. A. Dillow, *Lester, the lustre lister, version 00*, Dec. 2013. [Online]. Available: https://www.osti.gov//servlets/purl/1231806.

[24] M. Bruning, "Zfs on-disk data walk (or: Where's my data)," Prague, 2008. [Online]. Available: http://www.osdevcon.org/2008/files/osdevcon2008-max.pdf.

[25] D. I. Boomer, "Relational database active tablespace archives using hsm technology," in *23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2006), Information Retrieval from Very Large Storage Systems, CD-ROM, 15-18 May 2006, College Park, MD, USA*, 2006.

[26] J. LaFon, S. Misra, and J. Bringhurst, "On distributed file tree walk of parallel file systems," in *Proceedings of the 2012 International Conference on High Performance Computing, Networking, Storage, and Analysis (SC 12)*, 2012.

[27] *Libcircle*, https://github.com/hpc/libcircle.

[28] X. M. Zhang *et al.*, "Designing a sql query rewriter to enforce database row level security," Ph.D. dissertation, Massachusetts Institute of Technology, 2016.

[29] B. Yang, X. Ji, X. Ma, X. Wang, T. Zhang, X. Zhu, N. El-Sayed, H. Lan, Y. Yang, J. Zhai, W. Liu, and W. Xue, "End-to-end I/O monitoring on a leading supercomputer," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA: USENIX Association, Feb. 2019, pp. 379–394. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/yang.

[30] *Sqlite*, https://www.sqlite.org/index.html.

[31] D. Hitz, J. Lau, and M. Malcolm, "File system design for an nfs file server appliance," in *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, ser. WTEC'94, San Francisco, California: USENIX Association, 1994, p. 19.

[32] J. Bonwick, M. Ahrens, V. Henson, M. J. Maybee, and M. Shellenbaum, "The zettabyte file system," 2003.

[33] D. A. Dillow, *Lester, the lustre lister, version 00*, Dec. 2013. [Online]. Available: https://www.osti.gov//servlets/purl/1231806.

[34] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the xfs file system," in *Proceedings of the 1996 USENIX Annual Technical Conference (USENIX ATC 96)*, 1996, p. 1.

[35] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: Current status and future plans," in *Proceedings of the Linux symposium*, Citeseer, vol. 2, 2007, pp. 21–33.

[36] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," *ACM Trans. Storage*, vol. 9, no. 3, Aug. 2013. DOI: 10.1145/2501620.2501623. [Online]. Available: https://doi.org/10.1145/2501620.2501623.

[37] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in haystack: Facebook's photo storage," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10, Vancouver, BC, Canada: USENIX Association, 2010, 47–60.

[38] S. Morgan and M. Mortazavi. "Borgfs file system metadata index search." (2014), [Online]. Available: https://www.snia.org/educational-library/borgfs-file-system-metadata-index-search-2014.

[39] HPE. "Hpe data management framework." (2021), [Online]. Available: https://support.hpe.com/hpesc/public/docDisplay?docId=emr_na-a00025296en_us.

[40] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010. DOI: 10.1145/1773912.1773922.

[41] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, 56–65, Oct. 2016. DOI: 10.1145/2934664.

[42] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI 12)*, 2012, p. 2.

[43] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller, "Spyglass: Fast, scalable metadata search for large-scale storage systems," in *Proccedings of the 7th Conference on File and Storage Technologies (FAST 09)*, 2009, 153–166.

[44] H. Sim, Y. Kim, S. S. Vazhkudai, G. R. Vallée, S.-H. Lim, and A. R. Butt, "Tagit: An integrated indexing and search service for file systems," in *Proceedings of the 2017 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 17)*, 2017. DOI: 10.1145/3126908.3126929.

[45] H. Tang, S. Byna, B. Dong, J. Liu, and Q. Koziol, "Someta: Scalable object-centric metadata management for high performance computing," in *Proceedings of the 2017 IEEE International Conference on Cluster Computing (CLUSTER 17)*, 2017, pp. 359–369. DOI: 10.1109/CLUSTER.2017.53.

[46] M. Lawson, C. Ulmer, S. Mukherjee, G. Templet, J. Lofstead, S. Levy, P. Widener, and T. Kordenbrock, "Empress: Extensible metadata provider for extreme-scale scientific simulations," in *Proceedings of the 2Nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS 17)*, 2017, pp. 19–24. DOI: 10.1145/3149393.3149403.

[47] A. B. Keleher, K. Chard, I. Foster, I. Raicu, and A. Orhean, "Finding a needle in a field of haystacks: Metadata search for distributed research repositories," 2017.

[48] C. Gormley and Z. Tong, *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine*. " O'Reilly Media, Inc.", 2015.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We ran all of our experience on a CentOS 7 node running the Linux 5.7.9 Kernel. It has two Intel Xeon Platinum CPUs, 192 GB RAM, and 4 SSDs.

The software used were:

1. The fio benchmarking utility 2. blktrace 3. GUFI

Figure 7

Run fio on the target drives to get the available bandwidth. Create a tree on each of the different SSD configurations. Run gufi_query -E "SELECT uid FROM entries" <index> while running blktrace. Divide the block count by the query time to get the GUFI performance.

Figure 8

For each Rollup Limit, generate a new copy of the index from the same trace/tree and roll it up to the limit.

Run gufi_query -S "SELECT uid FROM summary" -E "SELECT uid FROM pentries" <index> to measure query performance of rolled up indicies.

Run du to get the sizes taken up by each index.

Figure 9

Replicate a fixed tree into 3 XFS directories. Set an xattr with a fixed name for 25%, 50%, and 100% of the non-directories in the indices, such as user.ext. For the same single file/link in each index, set the xattr to a fixed name such as "user.needle".

To benchmark XFS xattr performance, run find
${tree}
(-type l -o -type f
) | xargs -d '
n' -P 224 getfattr -h -n 'user.ext'

The tree used to obtain GUFI xattr performance was manually modified as xattr indexing was not completed at the time of writing. The 25%, 50%, and 100% xattr coverages were all placed into the same index under different table names, requiring only table name changes, which did not affect the overall performance of the experiments.

To obtain our scan results, we ran gufi_query -M 'rhentxattr,xrh% -E 'SELECT path(""), name, uid, gid, exattrs FROM pentries INNER JOIN myxatv ON inode=exinode" <index>

To obtain our stab results, we ran gufi_query -M 'rhentxattr,xrh% -E 'SELECT path(""), name, uid, gid, exattrs FROM pentries INNER JOIN myxatv ON inode=exinode WHERE exattrs LIKE "%needle%" <index>

Figure 10

We ran gufi_query with 4 different queries on a fixed tree as different users (sudo -u
<uid>).

1. gufi_query -E "SELECT name FROM pentries" <index> 2. gufi_query -S "SELECT name, size FROM summary" <index> 3. gufi_query -a -e 0 -I "CREATE TABLE sizes(total_size INT64)" -S "INSERT INTO sizes SELECT TOTAL(size) FROM summary" -E "INSERT INTO sizes SELECT TOTAL(size) FROM pentries" -J "INSERT INTO aggregate.sizes SELECT TOTAL(total_size) from sizes" -G "SELECT TOTAL(total_size) from sizes" <index> 4. After creating the tree summaries for each user with bfti, run gufi_query -T "SELECT size FROM tsummary" <index>

AUTHOR-CREATED OR MODIFIED ARTIFACTS:

Artifact 1

Persistent ID: `https://doi.org/10.5281/zenodo.6459552`
Artifact name: GUFI Github Repository
Citation of artifact: Jason Lee, Dominic Manno, Brad Settlemyer, & Gary Grider. (2022). mar-file-system/GUFI: XATTR (0.5.2-rc2). Zenodo. https://doi.org/10.5281/zenodo.6459552

Artifact 2

Persistent ID: `https://doi.org/10.5281/zenodo.6600686`
Artifact name: GUFI Filesystem Traces
Citation of artifact: Dominic Manno, Jason Lee, Brad Settlemyer, Qing Zheng, & Gary Grider. (2022). mar-file-system/GUFI-Filesystem-Traces: GUFI Filesystem Traces (v1.0.0) [Data set]. Zenodo. https://doi.org/10.5281/zenodo.6600687

Reproduction of the artifact with container: There is a CentOS 7 Docker image available at contrib/centos7-gufi.tar.gz in the code repository. GIT LFS is required to download it. The commands used to set up the environment and build GUFI are as follows:

on the host sudo docker run -it centos:7 bash

within the container # probably do not need sudo because you will be dropped into the container as root yum install -y automake epel-release gcc-c++ git make patch pcre-devel wget yum install -y cmake3

git clone -b sc22 https://github.com/mar-file-system/GUFI.git cd GUFI git checkout f36e7f856f99a32591d8d183a0d2c63059a28dd6 mkdir build cd build cmake3 .. make make install

————————————-

Publicly available traces of large trees can be downloaded from https://doi.org/10.5281/zenodo.6600686, which mirrors https://github.com/mar-file-system/GUFI-Filesystem-Traces. The repository contains ftp://hpc-ftp.lanl.gov/data/storage/GUFI/GUFITraces.tar.bz2 broken into pieces for easy upload/download with git lfs.

————————————-

To index a filesystem directory, run gufi_dir2index To convert a trace to an index, run gufi_trace2index To query an index, run gufi_query Argument examples: -S "SELECT * FROM summary" -E "SELECT * FROM pentries"

To roll up an index, run rollup on the index - the index will be modified. To roll up to a limit, run rollup with the -L <limit> flag. Querying a rolled up index requires no changes.

Use "-n <thread count>" with any of the GUFI executables to set the number of worker threads.