TagIt: An Integrated Indexing and Search Service for File Systems

Hyogi Sim[†,*], Youngjae Kim[‡], Sudharshan S. Vazhkudai[*], Geoffroy R. Vallée[*], Seung-Hwan Lim[*], and Ali R. Butt[†]
Virginia Tech[†], Oak Ridge National Laboratory[*], Sogang University[‡]
{hyogi,butta}@cs.vt.edu,youkim@sogang.ac.kr,{vazhkudaiss,valleegr,lims1}@ornl.gov

ABSTRACT

Data services such as search, discovery, and management in scalable distributed environments have traditionally been decoupled from the underlying file systems, and are often deployed using external databases and indexing services. However, modern data production rates, looming data movement costs, and the lack of metadata, entail revisiting the decoupled file system-data services design philosophy.

In this paper, we present TagIt, a scalable data management service framework aimed at scientific datasets, which is tightly integrated into a shared-nothing distributed file system. A key feature of TagIt is a scalable, distributed metadata indexing framework, using which we implement a flexible tagging capability to support data discovery. The tags can also be associated with an active operator, for pre-processing, filtering, or automatic metadata extraction, which we seamlessly offload to file servers in a load-aware fashion. Our evaluation shows that TagIt can expedite data search by up to 10× over the extant decoupled approach.

CCS CONCEPTS

• **Software and its engineering** → **File systems management**;
• **Information systems** → *Distributed storage*;

KEYWORDS

Distributed file systems, Search process, Indexing methods

ACM Reference Format:
Hyogi Sim[†,*], Youngjae Kim[‡], Sudharshan S. Vazhkudai[*], Geoffroy R. Vallée[*], Seung-Hwan Lim[*], and Ali R. Butt[†]. 2017. TagIt: An Integrated Indexing and Search Service for File Systems. In *Proceedings of SC17, Denver, CO, USA, November 12–17, 2017,* 12 pages.
https://doi.org/10.1145/3126908.3126929

1 INTRODUCTION

Big data management and analytics services play an ever crucial role in modern enterprise data processing, business intelligence, and scientific discovery. While the use of such services in the enterprise has received much of the attention, their use for scientific data analysis promises to produce the most impact. Consider scientific experimental facilities (e.g., Large Hadron Collidor [19], Spallation Neutron Source [15]), observational devices (e.g., Large Synoptic Survey Telescope [20]) and computing simulations of scientific phenomena (e.g., on supercomputers [18, 21]), which produce massive amounts of data that need to be analyzed for insights. For example, a 24-hour run of the fusion simulation, XGC [22], on the Titan machine [21] generates 1 PB of data each timestep, spread across O(100,000) files on the parallel file system (PFS), Spider [38]. The underlying storage system contains 1 billion files, and sifting through them to discover relevant data products of interest can be extremely cumbersome. Thus, there is a crucial need for fast and streamlined *data services* to search and discover scientific datasets at scale.

There are a number of well-established large-scale parallel and distributed file systems, such as GPFS [42], Lustre [8], HDFS [43], GlusterFS [17], Ceph [45], PanFS [46], PVFS [40], and GoogleFS [28]. However, these focus on scalable storage and failure resilience, but do not support the tight integration of scalable search and discovery semantics into the file system. While services such as indexing, searching and tagging exist for discovery in commodity, desktop file systems such as HFS+ [5] for Mac OS X or Google Desktop [4], such services cannot be simply extended or incorporated into PFS, especially at scale. Thus, many scientific communities still resort to manually organizing the files and directories with descriptive filenames, and use extensive file system crawling to locate data products of interest. Besides problems with scaling, such approaches lack the ability to capture more descriptive metadata about the data. This has led to ad hoc solutions and cumbersome approaches using manual annotations and domain-specific databases [1, 3]. Such solutions decouple the file system and the search/discovery infrastructure, where users explicitly publish the data products stored in the file system to an external catalog, and provide metadata, out of band of the data production process on the file system.

A number of factors underscore the need to revisit the decoupled philosophy for designing data services for scientific discovery. First, the decoupling of search/discovery from the file system inevitably results in inconsistencies between the data files and the external index. Second, since collecting metadata is a human-intensive process, oftentimes users only provide basic metadata during data publication to external catalogs, consequently limiting its efficacy. Instead, we argue that there is significant value in providing hooks so that users can annotate datasets in situ, as part of the file system. File systems already provide extended attributes as a way to add more metadata to files, which can be exploited to augment domain-specific information. Third, the dearth of metadata is only exacerbated by the rapid growth in data production rates and volume, and it can be very cumbersome for users to provide metadata about all of these data products in a post hoc fashion, i.e., (much) later than data production. There is a wealth of information buried within these files, which if harnessed efficiently can help answer

SC17, November 12–17, 2017, Denver, CO, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5114-0/17/11…$15.00
https://doi.org/10.1145/3126908.3126929

numerous data disposition questions. Fourth, growing data production rates imply that the data movement cost also grows manifold. Typically, the process of data analysis entails the discovery of relevant data or regions/variables of interest within the data, e.g., a variable within a netCDF [10] dataset, by posing a query to an external database catalog, and then moving the data from the file system to an analysis cluster for post processing. This process incurs a lot of unnecessary data movement. Instead, file system servers could potentially aid in such data reduction during the discovery process, thereby minimizing data movement. Finally, profiling of large-scale, production storage systems has shown that there are enough spare cycles on the file servers to take on additional services, e.g., Spider servers have been shown to experience less than 20% of their individual peak throughput for 95% of the time [30, 33]. While this may vary across deployments, there is the possibility of using the spare cycles for additional services.

Contributions

We present an integrated approach, TagIt, to address the above identified challenges. The goal of this project is to enable the indexing and search of data, resident on file systems, facilitating the fast and efficient discovery of data. Our design of TagIt integrates a data management service into the GlusterFS distributed file system [17], to support scalable indexing and search of scientific data.

Tagging Associating an index term or a “tag” to stored data for later quick retrieval has been shown to be very effective in commodity, desktop file systems [5], e.g., picture tagging, and improve productivity manifold. However, the underlying truly distributed architecture and scale requirements severely restrict the use of such systems in large-scale parallel and distributed file systems. TagIt extrapolates such capabilities to petabyte-scale file systems, wherein users can associate a richer context to collections of files by adding their own tags in order to quickly discover them, e.g., associating a piece of metadata, *“10th checkpoint of the Supernova explosion job run,”* to be able to quickly retrieve and operate on the tens of thousands of files from a job simulating Supernova explosions.

Distributed Metadata Indexing To realize the tagging functionality, we have designed a consistent and scalable metadata indexing service that indexes user-defined extended attributes, and is tightly integrated into a shared-nothing distributed file system. Hosting the metadata indexing service inside the file system effectively simplifies many consistency issues associated with the external database approach. The metadata index database is fully distributed across the available file system servers, each of which manages a horizontal shard of a global metadata index database for distributed query processing. The approach does not have any centralized components that can bottleneck in a large-scale deployment, and provides the needed scalability for complex queries, by evenly distributing the load to the available file system servers. Our scaling experiment indicates that TagIt scales to support large deployments, indexing over 105 million files from the production Spider PFS snapshot, using 96 logical volume servers (§ 5.2).

Active Operators We go beyond tagging to also support executing operations on tagged files. We have developed the ability to apply an operation or a filter on the file collections or specific portions of a file such as a stored variable (akin to marking a feature in a picture), which will be performed on the file system servers. This can be particularly useful when a user wishes to extract a large multi-dimensional variable, e.g., temperature, from a collection of files, upon which to run some analysis, e.g., mean temperature of an ice sheet dataset, instead of moving entire petabytes of data. Moreover, the user can also save and tag the results of such an *active operation*. This is similar to the ‘`find -exec`’ functionality, except that the operations are conducted on the file system servers, avoiding costly data transfers between the client and the file system. Our evaluation of the active operator feature on a large scientific dataset shows that it is very promising. For example, computing the decadal average for a large atmospheric measurement data collection (a 150 GB AMIP dataset with more than 130 files), used by the climate community, suggests that TagIt’s active operator can complete 10× faster than the traditional out-of-band calculation of the average, without having to move data to clients.

Automatically Extracting Metadata and Indexing To facilitate more sophisticated searches that can only be answered with richer metadata, a unique feature of TagIt is automatic extraction of metadata from files. Such operations can again be performed on the file system servers, but to reduce the impact on the servers, we limit them to a subset of files that the user deems worthy, e.g., a tagged collection. We can further index the extracted metadata similar to the user-defined metadata. Our results with the 150 GB AMIP climate dataset indicate that this advanced feature of extracting the metadata attributes (over 30 variables with arrays of values for each one) and indexing them only increases the index size on the file servers by 631 KB, suggesting the approach is very viable and can scale to larger collections of data efficiently.

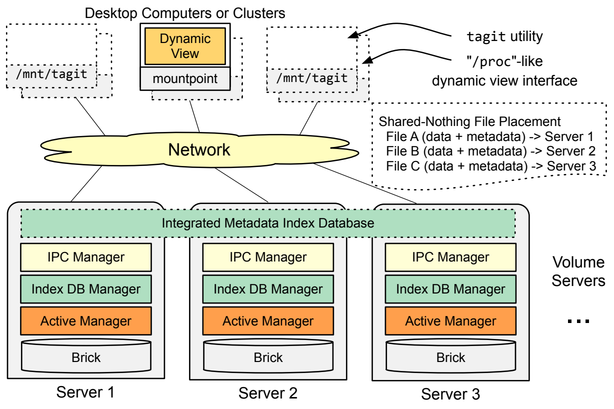

Desktop Computers or Clusters
/mnt/tagit
Dynamic View
mountpoint
/mnt/tagit
tagit utility
"/proc"-like dynamic view interface
Shared-Nothing File Placement
File A (data + metadata) -> Server 1
File B (data + metadata) -> Server 2
File C (data + metadata) -> Server 3
Network
Integrated Metadata Index Database
IPC Manager
Index DB Manager
Active Manager
Brick
Server 1
Server 2
Server 3
Volume Servers
...

Figure 1: Overview of TagIt architecture.

2 TAGIT OVERVIEW

The key design goals of TagIt are as follows: (i) Making file systems inherently searchable; (ii) enabling metadata capture; (iii) minimizing data movement; and (iv) building easy-to-use system tools and interfaces. In order to build a file system that natively supports scientific data discovery service, we have prototyped TagIt atop GlusterFS [17]. Particularly, GlusterFS features a shared-nothing architecture, which allows us to seamlessly integrate our ideas and demonstrate its efficacy in deployable systems.

Figure 1 presents the architecture of TagIt. Users can read and write data objects from the file system via a mount point. TagIt’s enabler GlusterFS is a shared-nothing [44] distributed file system. In GlusterFS, each backend file system is independent and self-contained. File metadata such as filenames, directories, access permissions, and layout are distributed and stored in backend file systems called *bricks*. Each brick is simply a directory inside a mounted file system (e.g., XFS). A logical *volume server* exports files inside a brick to clients. File metadata is stored in the same volume server as the associated file. This means that all operations to a single file are effectively isolated to a single volume server, obviating the need for centralized metadata servers.

In the above shared-nothing file system structure, we have integrated data management services within the volume server to manage the metadata index database, active operations for server-side data optimization, and metadata extraction. TagIt supports *tagging* of datasets using arbitrary user-defined file metadata that is internally stored as an extended attribute of the file. To facilitate search operations associated with such tags, TagIt internally indexes the tags and any metadata attributes about the datasets. The search index database of all attributes is tightly integrated into the file system itself, providing a strong consistency between the data file and the index. Moreover, the index is distributed across the volume servers, avoiding any centralized points, thereby achieving scalability. Beyond basic search, TagIt also supports *active operations*, which perform server-side data reduction or extraction to minimize data movement. Moreover, TagIt supports automatic metadata extraction to reduce laborious user annotation tasks. TagIt handles metadata extraction as an automatic active operation when processing data, and further indexes the extracted metadata for future search operations. Since such server-side processing can impact system performance, the automatic extraction is only done for datasets that the user has deemed worthy. Finally, *dynamic views* allow users to intuitively manage tags and active operators via virtual file system entries.

3 FILE SYSTEM-INTEGRATED METADATA INDEXING

In this section, we discuss the key building blocks of our approach, and how we build the metadata indexing mechanism and integrate it in GlusterFS.

3.1 Inverted Metadata Index Database

We adopt an *inverted index* data structure to facilitate efficient lookup of files in response to a search query. Inverted metadata index is used widely in Internet searches to identify pages that contain a particular search term. However, the approach has not been applied previously in the context of file system searching and querying at scale. Here, given a search term, we need to find collections of files with matching attributes.

Traditional file systems maintain file metadata in an inode, while the directory maintains a table of inodes to represent its files and sub-directories [37]. Thus, for any given pathname, the metadata is retrieved using a forward index structure. In our case, we wish to find a file collection, not only based on the pathname but also based on their metadata. The standard file system indexing structure is therefore not suitable for our needs, as it would require an exhaustive crawling of the entire file system, which is too costly with growing scale. By using an inverted index, our solution offers two major advantages: (i) enabling search queries based on user-defined attributes as well as system-defined attributes such as standard file system stat attributes, and (ii) avoiding crawling the file system. We implement the inverted index (henceforth referred to as index) using a relational database. We do not use key-value stores, as they are not well-suited for the lookup of multiple attributes from multiple tables at once, which is required by many practical file search operations (Table 2).

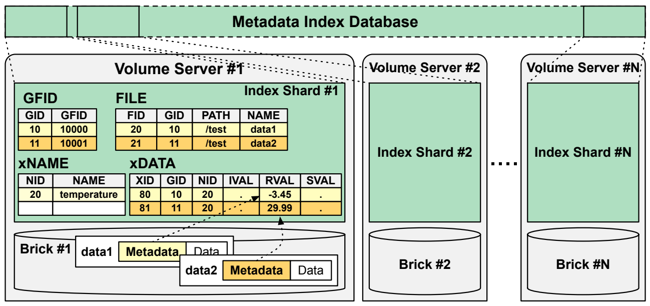


Figure 2: Sharded metadata index database in TagIt. Each index shard is tightly coupled with the local brick.

The relational schema is depicted in Figure 2. The index is implemented using four tables, GFID, FILE, xNAME, and xDATA. The database schema manages any user-defined attributes and system stat attributes in a unified way. File attributes are stored in two separated tables, xNAME and xDATA. For example, when a user assigns a new attribute, *temperature* as $-3.45°C$ and $29.99°C$ to files, data1 and data2, respectively, the attribute's name is added to the xNAME table. The attribute's value is added to the xDATA table, along with other necessary fields from GFID and FILE tables. Later, these files can be identified by performing a search based on the *temperature* attribute, e.g., find files with *temperature* < 0. The standard stat attributes are similarly stored (pre-populated) with pre-defined names in the xNAME table (*st_size*, *st_mode*, etc.).

3.2 Metadata Index Distribution

For the indexing service to support a large-scale file system, we need a *scalable* design, as well as *fault tolerance and durability* capabilities, i.e., fast recovery upon server failures and preventing server failure propagation. To this end, we split the metadata index into multiple *partitions*, so the load can be distributed between all the available volume servers. Note that the metadata index is deployed on existing file system servers, and not on additional servers. Practically, the metadata index database is horizontally divided into multiple partitions, and the partitions are scattered across the available volume servers. This horizontal partitioning is called database *sharding* [41], and each partition is referred to as a *shard*. With this architecture, each shard has its own (inverted) index database, i.e., its own table structure and search indices that are used to complete operations on database records (e.g., searching or updating records) independent of other shards. The database partitioning technique can effectively reduce the overall overhead associated with search operations by exploiting the multiple independent shards in parallel, as long as the records are *evenly* distributed across the shards.

Furthermore, as explained in § 2, operations on a file are limited to a single volume server that stores both the file data and metadata. In TagIt, we also provide this *shared-nothing* property to distribute all the records of the index database to the volume servers. Specifically, TagIt follows the file distribution algorithm of the underlying GlusterFS, i.e., each index database shard is populated with the metadata of files locally stored on the backend file system of the server (Figure 2). This tight coupling of the metadata index shard and the backend file system ensures that metadata and data are co-located, which has several benefits. Since files are uniformly distributed across volumes, the shards are also evenly distributed, effectively providing load balancing. Moreover, the shards catering only to their local volumes avoid any consistency issues across servers. Finally, the distribution of the shared index effectively isolates single server failures, simplifying the recovery process without affecting other servers in the cluster.

3.3 Synchronous Index Update

Solutions that are based on an index external to the file system require periodical *crawling* of the file system to keep the external database up-to-date and consistent with the file system. Solutions such as *change logs* to automatically capture file system updates have significant performance impact and thus are often not deployed on extreme-scale storage systems. Crawling entails the entire directory tree to be scanned and, for each file and directory, all of the attributes to be fetched, and is significantly slow. For instance, a crawling of the Spider file system [38], with about 32 PB and 1 billion files, takes over 20 hours. However, even with such a costly process, the records in the external index will get stale. To address this limitation, all file operations in TagIt trigger an update of the local index of the volume server, as part of the regular file system control path, rather than through an out-of-band mechanism. As a result, we refer to such an update as being *synchronous*. However, adding the extra burden of an index update to every file operation can substantially slow down the file system performance. In the following, we explain how TagIt is designed to minimize such a runtime impact.

Index Update TagIt index shards are updated upon every file operation that causes changes to the file system metadata. Such file operations include creating or deleting a file or a directory, changing attributes (e.g., changing the ownership or permission), and appending data to a file. All file operations in GlusterFS are implemented via I/O requests that are sent to the target volume servers, which use I/O threads to service the requests. We have added a synchronous update functionality to these threads. After completing the operation, an I/O thread checks whether the operation has changed any file attributes, and if so, updates the index shard accordingly. While the I/O thread is updating the index shard, it creates an UNDO log in memory, and exclusively modifies the index shard by acquiring an exclusive database lock. Such serialized database accesses affect the response time of all file operations, especially when thread concurrency for file operations increases. TagIt minimizes this overhead—associated with the critical section where multiple I/O threads wait for acquiring the database lock—by spawning a separate database update thread that exclusively updates the index shard. When an I/O thread needs to update the index shard, it creates and enqueues an "update request" to a shared queue. The database update thread continuously dispatches the update requests from the queue and applies the updates to the index shard. This design may introduce a slight latency, especially when a volume server is heavily loaded. We measured the latency by increasing the number of clients, each running heavy file and directory creation operations, and found it to be mostly negligible, e.g., under a millisecond for up to eight clients per a server (§ 5.1). Given the significant benefit our update approach provides for the foreground I/O operations, we argue that the delay offers a reasonable tradeoff.

Consistency As we discussed above, the asynchronous database update in TagIt may introduce a delay before an update request is dispatched and applied to the index database by the index update thread. For example, if metadata, X, is added to a file as an extended attribute, there may be a slight delay for the metadata to be propagated to the index database. Therefore, a search request for X could experience inconsistent results for a brief time. We chose the asynchronous update model due to its lower performance impact on file operations. However, for applications requiring stronger consistency, TagIt provides a command-line utility (`tagit-sync`) for ensuring all enqueued updates are promptly updated to guarantee consistent results, similar to `sync(1)` utility. The `tagit-sync` command provides stronger consistency while still minimizing the overhead for all file operations, by shifting the burden on the application requiring the higher consistency. Note that consistency of standard metadata read operations (e.g., `stat(2)`, `getxattr(2)`, etc.) is not affected by our asynchronous index update, since TagIt directly sends such operations to the backend file system.

Durability Changes to the database on index shard update should be written to the disk or SSD in order to survive unexpected server failures. However, triggering additional I/O operations for this purpose may decrease the overall server performance. Instead, each index shard is backed by a database file that is stored on the same backend file system, and the database file is mapped into memory (using `mmap(2)`) at runtime. As a result, TagIt does not trigger any extra I/O operations on database transaction commits, but instead relies on the periodic *dirty page sync* performed by the operating system. In other words, the consistency of the index shard only depends on the status of the backend file system; while this may lead to a loss of the index database records on server failures, TagIt can quickly recover any lost records as follows. Like most modern file systems, GlusterFS relies on a journal to track file system updates and prevent data loss. TagIt exploits the journal to avoid scanning the entire backend file system for identifying any missing updates in the index shard. After a server failure, the backend file system is recovered; then, TagIt detects the unclean shutdown and scans the journal in reverse order, looking for any missing updates in the index shard. For each missing file entry, TagIt fetches the associated metadata from the backend file system, by invoking `stat(2)`, `listxattr(2)` and `getxattr(2)` on each missing file in the recovered backend file system, and populates the index shard. The recovery process happens on a per volume server basis, since each index shard is only associated with a single backend file system on the same volume server.

3.4 Distributed Query Processing

TagIt processes a search query for a collection of files by broadcasting the query to all index shards. The communication overhead from the query broadcast increases in proportion to the number of servers; however, benefits from processing the query in parallel increases as the complexity of the query increases. Thus, TagIt can achieve significantly improved performance particularly for complicated file search queries, even with the query broadcasting overhead in a large-scale cluster, i.e., 96 volume servers (§ 5.2). However, for queries that require global aggregation and processing of results from the index shards (e.g., *top-k* queries), we will need further processing at the client to conduct a global analysis of individual query results. This is because, the shared-nothing model prohibits the volume servers from talking to each other. Finally, TagIt is expected to yield improved performance when the file search query is combined with further advanced actions, e.g., applying a specific operation to the resulting files, minimizing data movement.

4 TAGIT SERVICES

4.1 Service Architecture

We use the indexing mechanism to build advanced data services, such as *tagging*, *active operators*, and *dynamic views*. Tagging allows custom marking/grouping of files, and supporting it in petabyte-scale PFS has the potential to enable discovery of relevant data products from among hundreds of millions of files. Further, active operators (which run on the file servers) can be associated with the collections to minimize data movement between file servers and clients. The results can themselves be further tagged and indexed. These features allow for automatic metadata extraction as well. Finally, TagIt also supports virtual directories, where a user can associate a file search operation to a virtual directory for easy interactions and scripted operations on the selected files.

We have implemented a data management service framework inside the file system to support the above services. We also provide access to the services via a command-line utility, '`tagit`'. `tagit` relies on standard UNIX system calls, such as `setxattr(2)` and `getxattr(2)`. Figure 3 shows the service architecture of TagIt and its different software components. On the client side, data management requests triggered by `tagit` are sent to *IPC Managers* or *Dynamic View Managers* according to the type of the requested service. The *IPC Managers* handle communications between clients and servers through the GlusterFS translator framework [17], while *Dynamic View Managers* handle the *dynamic views*. On the server side, volume servers have both a *IPC Manager* for handling communications with clients, and an *Index DB Manager* for managing the local index shard. Furthermore, *Active Managers* execute the service side of the *active operators*. Finally, normal file I/O operations are handled through the *I/O Manager* provided by GlusterFS.

4.2 Data Management Services

Tagging Users can manage tags, e.g., create or delete a tag, using the `tagit` command, which in turn uses standard extended attribute operations (e.g., `setxattr(2)` and `removexattr(2)`) on the servers as needed, and *Index DB Manager* updates the index. Later on, such user-defined tags can be used in the context of a

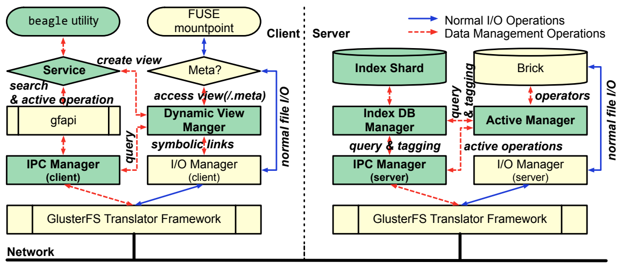
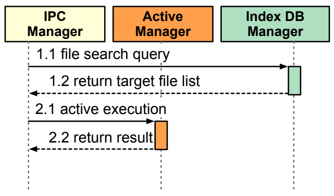


Figure 3: Service architecture of TagIt.

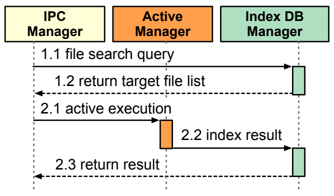
file search, together with other file attributes, e.g., name, size, etc. The restrictions for creating tags follow Linux's extended attribute policy. attribute in Linux VFS, For example, the size of each tag is limited by an extended attribute in the Linux VFS (i.e., 255 bytes for the name and 64KB for the value). The maximum number of extended attributes for a single file is file system-specific (e.g., unlimited in XFS), and the space consumption for storing extended attributes counts towards file system quotas. Therefore, even if a user deems too many files to be important, and creates tags, the enforced FS quota will prevent any overage.

Active Operators TagIt provides an easy interface for applying *operators* to a file collection of interest. The operators are run on the volume servers to avoid transferring data between clients and servers. Operators can be any user-specified commands, which are applied to file collections that results from a search query request. Suppose a user wants to run the `ncdump` program against all netCDF files in a directory, e.g., `/proj1`. The user executes the command '`tagit-execute /proj1 -name=*.nc -exec=ncdump`'. Upon execution of the command, the *IPC Manager* on the client broadcasts a request to all volume servers, which is similar to what happens when executing a file search. The *IPC Manager* on each server receives the search query and executes the request as it would do for a normal file search, but, instead of returning the results back to the client, the *Active Manager* executes the command (e.g., `ncdump`) on each file in the search result. The *Active Manager* also buffers the output of the command. When all active executions complete, the buffered output is returned to the client. Finally, the client receives the output from all the servers, combines them, and presents the output to the user. This sequence is depicted in Figure 4(a), and it is referred as $Operator_{simple}$.

TagIt also allows users to specify a *format-transformation* command as an operator (e.g., resizing image files or extracting a variable, *temperature*, from netCDF files) and run it against a set of searched files. Consider a search query, wherein a user wants to compute the average temperature of the monthly atmospheric measurement data (netCDF files) over a decade, from the Atmospheric Model Intercomparison Project (AMIP) experiment [27]. The files contain several properties (e.g., temperature, salinity) with their associated values that describe the experiment and are encoded in netCDF format. Let us further assume that the netCDF files have been tagged within TagIt, with the $AtmosphericMeasurement$ metadata. Our indexing of both the tags and the file system stat metadata will ensure that the netCDF files corresponding to the monthly atmospheric measurements over the last decade are quickly identified. However, without the ability to just extract the *temperature* variable (arrays of values) from the monthly data, and apply the



(a) $Operator_{Simple}$



(b) $Operator_{Advanced}$

Figure 4: Control flows of active operators inside a volume server.

mean function on the TagIt volume servers, we will need to move entire datasets to the client, which may contain other attributes such as salinity, etc. To address this, TagIt supports the format-transformation operator. This can be achieved by appending an extra argument specifying a directory, in which the transformed files will be stored. Internally, this works identically to $Operator_{simple}$, except that the *Active Manager* now creates an output file (in the specified directory) per execution: '`tagit-transform -outdir=/new -tag-id=dataset -tag-val=Measurement -exec=gettemp`'. The output files generated by the `gettemp` program will appear in the `/new` directory. This exploits the GlusterFS feature that each brick mirrors the entire directory tree but can also project newly created files in the local brick to clients. Only error codes from the runs are returned to the client. Note that the active operators in TagIt aim to reduce the data movement between the storage system and the client by providing a convenient framework for server-side data reduction. Applications may still need to perform additional operations, such as aggregation or sorting, to complete the analysis that requires extra communications, e.g., data shuffling.

We have extended $Operator_{simple}$ to interface it with the index services in order to provide more advanced capabilities. Suppose a user wants to extract the metadata of searched file collections, run the operators on them, and index the results after the operators are executed. For that, the user can specify the '`-index`' argument to the `tagit` command. In this context, the *Active Manager* buffers the output from each execution, as it does with a *Simple Execution*. However, in addition, each line of the output is parsed as a key-value pair (e.g., `dimension=5`) and the parsed pairs are tagged, i.e., added to the index shard and set as extended attributes to the input file(s). This process is depicted in Figure 4(b) and referred as $Operator_{advanced}$.

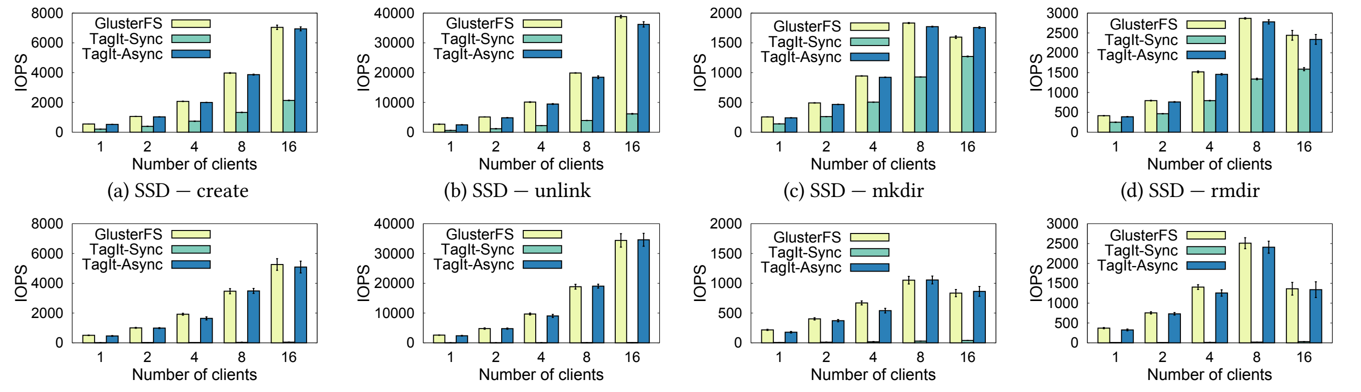
Security If users use active operators to execute untrusted binary code, the volume server can compromise the performance and security of the entire file system. To preclude malicious and buggy behaviors in untrusted user programs, the *IPC Manager* can manage a quarantined environment to run user supplied programs. Specifically, TagIt can adopt the Linux Container [7] for an isolation environment, and create an unprivileged container (i.e., lacking the superuser privileges) without any external network connections. We currently dedicate two CPU cores and 4GB memory to the container from a 12 core, 64GB volume server in our testbed (Table 1). Further exploration for building a secure environment is beyond the scope of this work.

Automatic Metadata Extraction Although TagIt can perform $Operator_{advanced}$ automatically for all the files in the file system, the sheer volume of data in extreme-scale file systems will overwhelm the file servers. Instead, TagIt allows users to trigger the automatic metadata extraction only for file collections that the user has deemed worthy. Specifically, a user can register a directory for automatic metadata extraction with an attribute such as '`tagit-autoindex /some/dir`'. After the directory is registered, TagIt automatically extracts metadata from all the files with specific file format extensions such as `hd5` and `nc` under the directory and indexes them. Internally, every volume server in TagIt maintains additional records of '`{extension, extractor}`' and the list of registered directories. When this feature is enabled, on every file close operation, TagIt additionally checks whether automatic extraction should be triggered. It is triggered only if the file is modified, the file has a known-type extension, and, lastly, one of the parent directories appears in the list of automatic extraction directories. If so, the file is enqueued to the extraction queue. An extraction helper thread (per volume server) applies the `extractor` program on the queued files.

The automatic metadata extraction framework also helps users keep the tags (or attributes) always up-to-date, i.e., consistent to associated data files. Specifically, if an attribute P has been extracted from a file F via the automatic metadata extraction framework, P becomes inconsistent if the contents of F change. TagIt has an elegant way to address this by virtue of the registration mechanism outlined above. Since users need to register a directory for TagIt to automatically extract the metadata, whenever the contents of the file F change, TagIt will rerun the `extractor` program and update P. As a result, the contents of the file F and the associated attribute P will remain consistent without any user intervention.

Dynamic Views A dynamic view provides a way to the users to save their search queries, and is created with the `tagit` command by passing an additional '`-create-view`' argument and a view name, for any file search request. Upon receiving the request, the *Dynamic View Manager* writes the dynamic view information to a temporary data file, *view list*. The view list file is local to the client, i.e., maintained on a per-client basis. After its creation, a new virtual directory appears under `/.meta/views`. The `/.meta` is a root of the virtual entries (i.e., temporarily existing only in memory) in GlusterFS, similar to `/proc` in Linux. Each time a user reads the `/.meta/views` directory, the *Dynamic View Manager* dynamically generates directory entries based on the view list file. Also, each directory entry is associated with a file search query that is specified during the creation of a view. Correspondingly, when a user reads a particular dynamic view directory, the *Dynamic View Manager* performs the distributed query through the *IPC Managers*. With the result of the query (list of files), the *Dynamic View Manager* creates symbolic links pointing to the search result files. This process, and dynamically generating directory and symbolic links, happen solely on the client, without burdening the file servers. Further, all directories and symbolic links under `/.meta/views` are transient, without occupying any memory or disk space when they are not accessed. Note that the dynamic view is similar to *Views* in a relational database [25]. In fact, the dynamic view in TagIt can be seen as a database view that is externally managed and wrapped by a file system interface.

Although, a dynamic view only exists temporarily by default on a single client, there exist cases in which certain views may need to be kept permanently and globally (e.g., sharing the view between multiple clients). In TagIt, users can create *permanent dynamic views*, and make an existing dynamic view permanent



(a) SSD — create (b) SSD — unlink (c) SSD — mkdir (d) SSD — rmdir

(e) HDD — create (f) HDD — unlink (g) HDD — mkdir (h) HDD — rmdir

Figure 5: Performance overhead of metadata indexing in the file system. `mdtest` [9] benchmark was used to generate metadata-intensive workloads. We used two different storage volume configurations, with SSDs ((a)–(d)) and with HDDs ((e)–(h)), to observe the performance impact of storage device characteristics.

as well. All permanent views appear globally on all clients. This is achieved by keeping the list of the permanent dynamic views using a special hidden file (`/._views`) inside the file system. The permanent dynamic views appear under `/.meta/views/sticky`, and are handled similarly as the (temporary) dynamic views. Note that, in this context, a client only needs to fetch view names and search queries from the file server upon the execution of a user request. Once the name and search query of a permanent dynamic view are acquired, all directories and symbolic links are processed on a single client as for the (temporary) dynamic views.

4.3 Discussion

The techniques used in TagIt are applicable to other PFS such as Lustre [8] and Ceph [45], with appropriate modifications. TagIt mainly requires modest computational resources on the PFS servers to run the lightweight database shards and active operators. For example, Ceph supports a key-value store, RocksDB [12], for supporting atomic object writes, which TagIt can use for indexing and other operations. Similarly, basic tagging can be supported as before. One consideration is that PFS with centralized servers already suffer from performance bottlenecks (e.g., Lustre, which is moving to multi-server DNE [2]). Thus, advanced TagIt services such as indexing of the tags can (should) only be run on PFS with multiple, distributed metadata servers that can handle the extra load. Finally, to support active operations for striped files, e.g., on Lustre or GPFS [42], we will need to aggregate the stripes from the backend servers. This requires additional communication and data movement between the servers, and may impact performance.

5 EVALUATION

Implementation TagIt has been implemented atop GlusterFS 3.7, an open-source distributed file system. We extended the translator framework in GlusterFS to implement index database services (index shard) and science discovery services (active operator and dynamic views). On the server side, an index shard translator is implemented using a light-weight database, SQLite [16]. On the client side, dynamic views are implemented in the *meta* translator, a virtual file system framework in GlusterFS. TagIt command-line utilities are implemented using the GlusterFS library (*glapi*). For evaluating TagIt, we consider two implementations—TagIt-Sync and TagIt-Async. In TagIt-Sync, the index database is synchronously updated, while in TagIt-Async, a dedicated thread is spawned to update the database asynchronously (§ 3.3).

Testbed Table 1 shows our testbed, where we used a private testbed with 32 nodes connected via 1 Gbps Ethernet, configured as 16 servers and 16 clients. For a realistic performance comparison, we used both synthetic and real-world workloads. For synthetic workloads, we used `mdtest` [9] and `IOR` [6] benchmarks for file metadata and file I/O intensive workloads, respectively. For a real workload, we used real-world scientific datasets such as the AMIP atmospheric measurement datasets [27]. All experiments were repeated six times, unless otherwise noted, and we report an average with a 95% confidence interval.

	Server (16)	**Client (16)**
CPU	12-core Intel Xeon E5-2609	8-core Intel Xeon E5-2603
RAM	64 GB	64 GB
OS	RHEL 6.5 (Linux-3.1.22)	RHEL 6.5 (Linux-3.1.22)
Network	1 Gbps Ethernet	1 Gbps Ethernet
Storage	Intel 240 GB SSD, Seagate 1 TB HDD	N/A

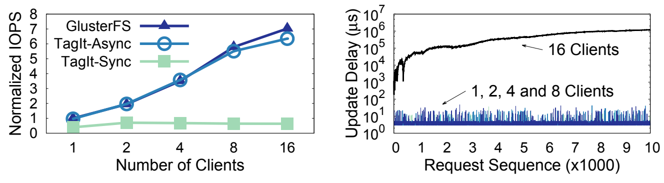
Table 1: Testbed specification.

5.1 Metadata Indexing Overhead

In our first test, we study the performance overhead of the integrated index databases of TagIt on the GlusterFS volume servers, while servicing file I/O operations.

Metadata-Intensive Workloads Figure 5 shows the performance comparison of TagIt and GlusterFS for metadata-intensive workloads, including file operations (e.g., create and unlink) and directory operations (e.g., mkdir and rmdir). We increase the number of clients from 1 to 16. In order to see the impact of the storage device characteristics, we considered both SSD and HDD volume server configurations.

Figures 5 (a)–(d) show the results with the SSD volume configuration. In file operations (Figure 5 (a)–(b)), we see that both TagIt and GlusterFS scale linearly with respect to the number of clients. Further, we can see that the throughput of TagIt-Async is only



(a) Normalized throughput (b) Queue delay

Figure 6: Experiments with an overloaded server. (a) shows the normalized throughput, and (b) depicts queueing delays of database update requests.

4% lower than the throughput of GlusterFS, on average. However, TagIt-Sync exhibits a noticeably decreased throughput compared to GlusterFS, due to frequent database file sync operations. For directory operations (Figures 5 (c)–(d)), TagIt-Async and GlusterFS scale only up to 8 clients. This can be attributed to the fundamental design of GlusterFS, in which all directories are replicated in every volume server (§ 2). Figures 5 (e)–(h) show the results with the HDD volume configuration. Not surprisingly, we have similar observations as in Figures 5 (a)–(d), except that the throughput under TagIt-Sync are too low to be discernible in the graphs.

Impact of Server Congestion The preceding experiments were conducted with the number of clients being less than or equal to the number of servers. In our next test, we consider a case in which servers are overloaded by more clients. To create the overloaded condition, we increased the number of clients from 1 to 16 while keeping a single server. Each client concurrently creates 10,000 files in its own directory.

In Figure 6 (a), we observe that TagIt-Sync does not scale with more than four clients. In contrast, TagIt-Async scales similarly to GlusterFS. However, with 16 clients, we notice TagIt-Async shows lower throughput than GlusterFS. This is because the database update thread in TagIt (§ 3.3) is overloaded and cannot keep up with the speed of incoming requests. This can introduce a non-negligible delay for updating the database, which in turn may result in an inconsistency between the file system and the index database (§ 3.3). To investigate the delay, we measured database update latencies of the first 10,000 create requests. Figure 6 (b) presents the delays with respect to the request sequence in time-series. We observe that, for up to eight concurrent clients, the delays are under 1 millisecond for all requests. However, the delay increases up to above 20 seconds with 16 clients. Overall, TagIt-Async performs similar to GlusterFS, and it is important to properly estimate the maximum server load to keep the metadata index database consistent.

I/O Intensive Workloads Figure 7 shows the performance overhead of metadata indexing for representative I/O patterns for scientific applications. In specific, we perform our tests for both a single shared file I/O model (N processes reading and writing to a single file, N1-Read and N1-Write in the figure) and a per-process file I/O model (N processes reading and writing N files, NN-Read and NN-Write in the figure). For the N1 tests, a single shared file is created for 16 clients, and each client concurrently appends 4 MB at a time until the aggregate size of file operations per client reaches 1 GB (16 GB total). For NN tests, each client writes in its own file separately. Overall, for both tests, we see little performance degradation due to the metadata indexing in TagIt.

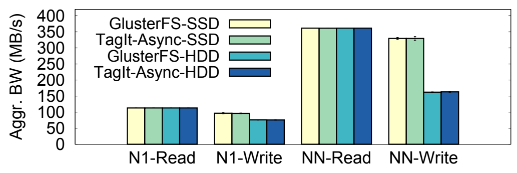


Figure 7: Performance comparison of GlusterFS and TagIt-Async for parallel I/O workloads. `IOR` benchmark [6] was used to generate N1 and NN workdloads.

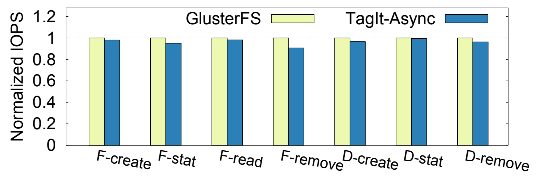


Figure 8: Metadata indexing overhead of TagIt for a large deployment. *F-* and *D-* denote the file and directory operations, respectively.

Crash Recovery TagIt recovers from a server failure by repopulating any lost updates to the index database. From a single server failure, the recovery program of TagIt can recover 351.95 files per second, e.g., for the lost metadata updates of 10,000 files, TagIt can repopulate the local index shard within 30 seconds.

Indexing Overhead at Scale Here, we evaluate the performance of TagIt on a large cluster to study how TagIt performance scales with an increased number of volume servers and clients. The testbed cluster consists of 104 diskless nodes, each of which is equipped with two four-core Intel Xeon E5410 processors (total eight cores) and 16 GB RAM. The nodes are connected via an infiniband network (Mellanox MT25208, 10Gbit/sec). We configured the file systems (GlusterFS and TagIt-Async) with 80 volume servers using 80 physical nodes. A memory file system (*tmpfs*) was used as a backend storage on the volume servers. The rest of the 24 nodes were used as clients. To evaluate the metadata indexing overhead, we ran the `mdtest` benchmark by spawning two processes on each client node (total 48 client processes). Figure 8 shows the result with seven different metadata operations, namely create, stat, read, and remove (unlink) for files and directories (with the exception of reads for directories). F- and D- denote file and directory operations, respectively. Each test was run five times, and since there was very little variation between the runs, we only present the average. For each operation, the TagIt-Async throughput is normalized to the GlusterFS throughput. We observe that the indexing overhead of TagIt is less than 5% in all cases, except for the file remove operation (F-remove) where the overhead is around 10%. Since file remove (unlink) is the fastest metadata operation in GlusterFS (Figure 5), its indexing overhead is more discernible than other operations. Overall, this result is consistent with our previous observation, and the indexing overhead of TagIt is not affected by the cluster scale due to the shared-nothing architecture.

5.2 File Search Performance

In our next tests, we evaluate the effectiveness of file searches in TagIt compared to an external database approach. Since SQLite does not support the server mode, 16 MySQL servers (identical to the number of volume servers in TagIt) are used to evaluate the external

	Description	Attributes	Tables	Results (#)
Q1	Locate files and directories with pathname containing '`never-existing`'.	name	FILE	0
Q2	Count the number of all regular files under '`/proj`', owned by a user.	path, mode, uid	FILE, xNAME, xDATA	1
Q3	Find regular files with a '`.mpi`' extension owned by a group, under '`/proj`'.	path, name, mode, gid	FILE, xNAME, xDATA	3
Q4	List all files owned by a group.	path, mode, gid	FILE, xNAME, xDATA	647
Q5	List all regular files which have been created in the last 24 hours.	path, mode, ctime	FILE, xNAME, xDATA	50,552

Table 2: Various file search queries to measure the query performance. Attribute column shows metadata required to answer the query, while table column shows database tables that hold the metadata columns.

Number of Clients		**1**		**2**		**4**		**8**		**16**	
System		**MySQL**	**TagIt**	**MySQL**	**TagIt**	**MySQL**	**TagIt**	**MySQL**	**TagIt**	**MySQL**	**TagIt**
Q1	Total Runtime (s)	2.780	**0.840**	3.716	**1.580**	7.689	**3.026**	19.659	**5.843**	41.846	**11.392**
	Avg. Latency (s)	0.043	0.016	0.050	0.018	0.074	0.033	0.087	0.061	0.154	0.152
	95th Percentile	0.056	0.018	0.085	0.033	0.175	0.063	0.424	0.124	0.866	0.249
	99th Percentile	0.059	0.024	0.086	0.035	0.191	0.064	0.429	0.125	0.875	0.250
Q2	Total Runtime (s)	15.499	**6.840**	68.599	**13.471**	165.501	**26.408**	401.340	**53.409**	815.478	**103.839**
	Avg. Latency (s)	0.306	0.131	1.202	0.164	0.909	0.292	1.640	0.542	9.885	1.192
	95th Percentile	0.309	0.136	1.366	0.268	2.809	0.530	6.167	1.075	16.043	2.125
	99th Percentile	0.311	0.158	1.398	0.272	4.122	0.542	11.034	1.079	16.654	2.160
Q3	Total Runtime (s)	**6.052**	12.927	**6.918**	25.537	**8.783**	51.731	**17.759**	98.743	**38.110**	190.289
	Avg. Latency (s)	0.032	0.064	0.034	0.097	0.038	0.169	0.051	0.216	0.077	0.613
	95th Percentile	0.121	0.257	0.132	0.508	0.171	1.027	0.347	2.041	0.736	3.843
	99th Percentile	0.121	0.259	0.146	0.520	0.183	1.056	0.368	2.099	0.783	4.108
Q4	Total Runtime (s)	16.711	**8.508**	67.476	**16.278**	161.971	**32.474**	409.635	**64.828**	795.376	**131.545**
	Avg. Latency (s)	0.320	0.163	1.185	0.206	0.987	0.293	1.428	0.855	7.776	1.632
	95th Percentile	0.325	0.168	1.339	0.318	2.724	0.635	6.044	1.427	15.646	2.691
	99th Percentile	0.356	0.195	1.388	0.329	4.097	0.819	11.183	1.710	16.258	3.277
Q5	Total Runtime (s)	**32.390**	49.420	128.516	**50.727**	326.109	**76.295**	803.266	**153.888**	1512.220	**312.241**
	Avg. Latency (s)	0.387	0.703	1.329	0.701	1.127	1.106	1.691	1.868	9.247	3.594
	95th Percentile	0.647	0.905	2.525	0.912	5.540	1.603	10.832	2.949	29.763	6.089
	99th Percentile	0.649	0.917	2.664	0.953	7.589	1.756	22.368	3.398	30.898	6.484

Table 3: Query performance under TagIt vs. the crawling approach with 16 MySQL servers.

database approach. Note that, in TagIt, such external servers are not needed, because the database is integrated into the file system. We used the same server machines with SSDs for both cases (Table 1), and all SSDs were formatted with the XFS file system. For a realistic workload, we used a snapshot of the Spider file system [38], taken on July 1, 2015. The snapshot contains information on pathnames and attributes of 1,303,156 files and 3,294 directories.

Index Database Population Overhead TagIt populates index shards during file operations, whereas the external database approach has to perform a periodic update. Specifically, the external database approach requires the following steps. First, the entire file system has to be scanned to generate a current file system snapshot. Second, databases are populated with the file system snapshot. In our experiment, we developed an in-house program to take a file system snapshot using `find` and `stat` system utilities and populate the databases, although the scanning process could be expedited [34]. The 16 MySQL servers of the external approach were populated in parallel from 16 clients.

Table 4 compares database management overheads for TagIt and the external database approach in terms of database space and update overheads. Both approaches use the similar amount of storage space for storing the databases. Specifically in TagIt, the index shard per server only requires 110.63 MB. To build its database, the external database approach takes about 96 minutes to populate the index; 93 minutes to crawl the file system and generate a file system snapshot, and about 4 minutes to update the 16 MySQL servers. Although the database population process could overlap with the file system crawling process, its improvement

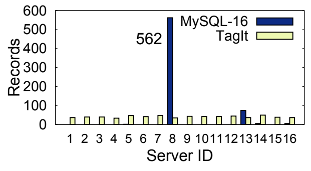
	MySQL-16	**TagIt**
Database Size	1971.39 MB	1770.08 MB
Crawling/Update Time	96.10 min	N/A

Table 4: Database size and update time under TagIt vs. the crawling approach with 16 MySQL servers.

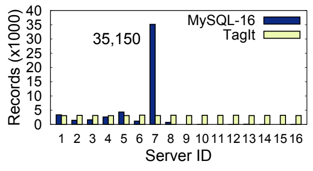
would be minimal because the file system crawling time is dominant in the entire database population time. Such long delays can lead to inconsistency between the file system and the database and are clearly undesirable, especially in large-scale file systems.

File Search Performance To compare the file search performance, we used the databases that have been populated in the previous experiment, and tested with five realistic stat-based queries for file searches as shown in Table 2. Note that these tests are also representative of tagging-based file searches. To measure the query performance, we wrote a C program that repeatedly executes a given SQL query 50 times. To test a multi-user environment, we measured the performance by increasing the number of clients to 16. We also used a warm-up period of a minute for each query test. Table 3 shows the total runtime and the summary of individual database request latencies for each case. We observe that TagIt can process Q1 query about three times faster than MySQL. Note that Q1 is a simple query that requires a full scan of an entire column without resorting the database index. In our experiments, SQLite could process this type of query faster than MySQL. For Q2, Q4, and Q5, TagIt also outperforms MySQL. We see that TagIt outperforms MySQL by a factor of 7, when using 8 or more clients.

In order to further investigate the lower query performance of MySQL for Q2, Q4, and Q5, we analyzed the query load distribution



(a) Q4 — 647 records



(b) Q5 — 50,552 records

Figure 9: Distributions of records for MySQL and TagIt. The record distribution of Q2 is similar but we do not show the result due to the page limitation.

Systems	r(Q1)	r(Q2)	r(Q3)	r(Q4)	r(Q5)
MySQL	2.678	53.638	2.188	52.456	99.731
TagIt	0.702	6.475	11.790	8.211	18.139

Table 5: Coefficients of linear runtime functions with the number of clients as an explanatory variable. In all cases, R^2 values are greater than 99%.

across servers. In particular, we counted the number of processed result records of each query in all MySQL servers. Surprisingly, we found that MySQL exhibits a heavily skewed distribution of the result records across servers for these queries (Q2, Q4 and Q5), as shown in Figure 9. In Figure 9, we can clearly see that there is a severe load imbalance across the 16 MySQL servers in the external database approach. For Q4, 562 records (total 647) are processed on a single server, and similarly for Q5, 35,150 result records (total 50,552) are processed on a single server. Moreover, for Q2, a single server had all matching 124 records. The reason for this heavily skewed record distribution can be attributed to the way that the databases are populated. In the external database approach, records are distributed based on the order in which they appear in the snapshot file. The snapshot file is created by crawling the file system tree, and files in the same directory are likely to appear continuously. In contrast, TagIt evenly distributes the records to all 16 volume servers because the distribution of the records follows the file distribution policy of GlusterFS, i.e., a distributed hash table.

Such a skewed distribution of records not only negates the benefit of the parallel query processing, but also significantly slows down the overall processing time. Note that a single query processing internally involves communication with all 16 database servers due to the nature of the sharded database architecture. Thus, a query cannot be answered until the slowest server completes its processing. We can observe this problem in Table 3, particularly by comparing average latencies with 95th and 99th percentile latencies. For instance, in MySQL with 8 clients, 99th percentile latencies are 6.7×, 7.8× and 13.2× higher than the average latencies for Q2, Q4 and Q5, respectively. For Q3, MySQL processes faster than TagIt. It is because MySQL can prune the result record set based on the file name (‘`%.mpi`’) prior to other conditions, which alleviates the negative impact of the skewed record distribution.

We also compared the scalability of query processing performance under increasing number of clients. For a fair analysis, we used a simple linear regression with the runtime measurements in Table 3. We compared the slope of the fit line for each query. Table 5 shows the coefficient (r), the slope of the fit line, of the runtime function with the number of clients as an explanatory variable. Note that a higher r value implies that the runtime increases more

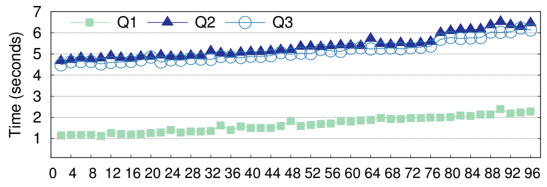


Figure 10: Query performance scaling under increasing volume servers and 105 million files.

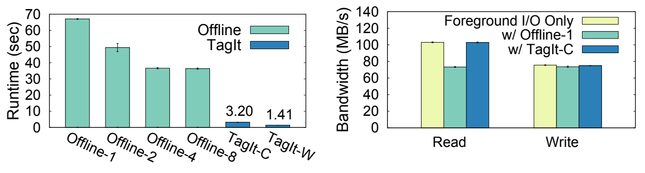
sharply as the number of clients increases. We observe that for Q1, TagIt and MySQL have similar slopes, however for Q2, Q4 and Q5, MySQL shows much higher slopes than TagIt, implying that MySQL scales worse than TagIt. For Q3, we see that MySQL scales better than TagIt.

Search Performance at Scale Next, we evaluate the overhead of query broadcasting (§ 3.4). In particular, we build the file system with 96 volume servers, and populate them with 105 million files from the Spider II snapshot file. We perform this experiment using 48 nodes of the Rhea cluster at Oak Ridge Leadership Computing Facility [11]. After populating the file system, the overall database size is 140 GB ($\mu = 1.45$ and $\sigma = 0.07$ across 96 volume servers). We execute Q1, Q2, and Q3 in Table 2 from a single client while varying the number of volume servers from 2 to 96. Note that for Q3, the number of resulting records is 4,766 in this setup. Figure 10 shows the result. We observe that executing Q2 and Q3 takes substantially longer than Q1, mainly because of the difference in the complexity of the queries. Q1 only needs to scan a single column (path) from a single table, whereas Q2 and Q3 require scanning and joining multiple database tables. In addition, for all queries, the benefit of sharded architecture outweighs the overhead of broadcasting. Using linear regression, we find that adding a single volume server merely increases runtime by 0.013×, 0.018×, and 0.016× for Q1, Q2, and Q3, respectively. For instance, executing Q3 with 96 volume servers takes 6.1 seconds, which is only 1.6 seconds more than the runtime with two volume servers (4.5 seconds).

5.3 Science Discovery Services

Evaluation of Operator$_{\text{simple}}$ To study the effectiveness of active operators, we used the query of computing the decadal average temperature of the AMIP atmospheric measurement datasets, composed of 132 1.2 GB netCDF files (total 150 GB) (§ 4.2). We wrote a dedicated program (operator), using the netCDF library, which calculates an average of the temperature variables in a netCDF file. We execute the program using two different methods, *Offline* and TagIt. In *Offline*, the program is run on a client and reads files from the file system. In TagIt, we offload the execution of the program using the operator framework. In *Offline*, we increase the number of threads from 1 to 8 to observe the impact of parallelism. We also evaluated the impact on the performance of normal I/O operations when they are performed during the program executions.

Figure 11(a) shows the results without any foreground I/O. We observe that for *Offline*, the run time decreases as we increase the parallelism. However, this happens only up to 4 clients. With 8 clients, the effect of parallelism almost disappears because of the I/O contention between the threads. In contrast, we see that TagIt performs noticeably faster than *Offline*. Note that TagIt not only utilizes multiple file servers to run the operators, but also performs



(a) Without foreground I/O (b) With foregound I/O

Figure 11: Performance impact of active operators in TagIt. (a) Performance under Offline vs. TagIt. (b) Impact on foreground I/O operations. TagIt-C and TagIt-W show the cold and warm volume server cache case, respectively.

near-data processing, minimizing data movement between the file servers and the client. Moreover, due to the shared nothing architecture of TagIt, there is little I/O contention between the operators running on the different servers. Figure 11(b) shows the results when either *Offline-1* (one thread) or *TagIt-C* runs concurrently with a foreground I/O operation. To understand the impact from overlapped executions, we launch a separate client that either reads or writes a 1 GB file sequentially. Under the read workload with *Offline-1*, the I/O bandwidth drops by about 30%. However, under the write workload, there is little impact on the foreground I/O both from *Offline-1* and *TagIt-C*. This is because the foreground write operations are cached by the client before reaching the servers, and are not directly affected by the server-side contentions.

Evaluation of Operator$_{\textbf{advanced}}$ Next, we evaluate the use of active operations to extract and index the metadata from scientific data (e.g., netCDF). We study the performance impact of performing the additional indexing on the file servers. Specifically, we compare the performance of the following two cases. In the $Operator_{simple}$ case, the file server executes a program that calculates a statistical summary (min, max, mean, median, etc.) from a netCDF file. In the $Operator_{advanced}$ case, the file server executes the same program, but the result is also indexed as attributes of the netCDF file. This involves setting extended attributes and adding records to the index shard. We used the same AMIP dataset (132 netCDF files, 150 GB) as before. Despite the additional processing on the file servers, $Operator_{advanced}$ runs 10% faster (1.45 s vs 1.65 s on average across 6 runs) than $Operator_{simple}$. This is because processing the results locally on the file servers is faster than gathering all results on the client. Note that, in the $Operator_{simple}$ case, the raw results are not processed further, but are simply aggregated and displayed to the user on the client.

In $Operator_{advanced}$, indexing the extracted metadata from the AMIP datasets increases the index database size. The raw data size of the extracted metadata (31 attributes) from a single netCDF file is about 1.5 KB and, with 132 netCDF files, the total database size increases by 631 KB on 16 volume servers. That is, each netCDF file increases the size of the index database by only about 3.2 KB. For a larger scale test, consider the project directory snapshot (1.3 million files) from the Spider file system used in § 5.2, which includes 787 complex files (631 netCDF, 180 FIT, and 4 HDF5 files). Suppose that, for this experiment, all such files are indexed after extracting 31 metadata attributes. Then, the total index database size will increase only by up to 2518.4 KB (787 × 3.2), or 157.4 KB per index database shard, compared to the original database size (1770.08 MB, refer to Table 4). While this is promising, it is also dependent on the data collections and their metadata content. Therefore, we will need to be judicious, and only extract and index metadata for data that the user deems important.

6 RELATED WORK

Managing metadata in a large-scale file system has been the focus of many works. GIGA+ [39] is a directory service that can be stacked on any parallel file systems. FusionFS [49] employs a distributed key-value store for a scalable metadata management. Recently, DAOS [35] proposes a new parallel file system architecture based on a distributed object-based storage, to address the limitations of the traditional POSIX interface in emerging extreme-scale platforms. Although these systems are scalable and alleviate the metadata overhead of file systems, unlike TagIt, they do not directly implement *searchability* that requires further indexing and management of metadata, as we have previously explained in § 3.1.

File system searchability has mostly been achieved by using external applications in a post hoc fashion [4, 36]. However, keeping the search index up-to-date with graceful performance degradation is non-trivial even in a single-user system [23]. The research community generally anticipates magnified challenges for maintaining a search index for large scale file systems. Spyglass [34] reduces the crawling overhead, but the solution is specialized to the architecture of the NetApp WAFL file system [31]. In contrast, TagIt addresses such shortcomings and provides a scalable data management service. VSFS [47] offers a searchable FUSE-based file system interface that sits on other parallel file systems, and provides a namespace-based file query language, similar to Semantic File System [29]. However, VSFS still maintains a metadata index outside of the file system, and thus requires its own data distribution and servers to scale [48]. The integrated design of TagIt precludes such extra servers and custom distributions. HP StoreAll Express-Query [32] is a production archival storage system that provides a rich metadata service, using a distributed database [26]. As before, the use of a decoupled metadata database is a limiting factor in this system as well. Moreover, these systems do not support advanced data management services (§ 4). Apache Lucene/Solr [14] supports automatic metadata extraction for well-known file types. However, the system also requires file system crawling due to its decoupled architecture. SciDB [13] is a database system specialized for scientific applications, and provides pre-processing of datasets, such as transporting a vector-based dataset. DataHub [24] offers `github`-inspired scientific data management and sharing, based on database techniques. However, both designs require using a custom interface instead of a file system, which creates an unnecessary and impractical hassle for users. In contrast, TagIt provides both searchability and pre-processing within the file system via the familiar command line interface.

7 CONCLUSION

In this paper, we have presented a case for tightly integrating data management services within file systems to enable rich search semantics therein. Traditionally, such services are provided via database catalogs external to the file system, which is not sustainable in the face of emerging data generation trends. TagIt maintains a scalable and consistent metadata index database inside the file system and offers advanced data management services including tagging, search, and active operations, to expedite scientific discovery processes. TagIt also features an easy-to-use user-interface; a dedicated command line utility provides similar semantics of the traditional `find` utility, and the dynamic view organizes data collections of interests in an intuitive directory hierarchy. Our evaluation with TagIt implemented atop GlusterFS shows that TagIt is viable and outperforms an external data management approach, without the need for deploying any additional resources.

Acknowledgement

We would like to thank our shepherd, Suzanne McIntosh, for her feedback. This research was supported in part by the U.S. DOE's Scientific data management program, by NSF through grants CNS-1615411, CNS-1405697 and CNS-1565314, and by the Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea Government (MSIP) (No. R0190-15-2012). The work was also supported by, and used the resources of, the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at ORNL, which is managed by UT Battelle, LLC for the U.S. DOE (under the contract No. DE-AC05-00OR22725).

REFERENCES

[1] *ARM Climate Research Facility.* http://www.arm.gov/.
[2] *DNE 1 Remote Directories High Level Design - HPDD Community Space - HPDD Community Wiki.* https://wiki.hpdd.intel.com/display/PUB/DNE+1+Remote+Directories+High+Level+Design.
[3] *ESGF, Earth System Grid Federation.* http://esg.ccs.ornl.gov.
[4] *Google Desktop.* http://desktop.google.com.
[5] *HFS Plus - Wikipedia, the free encyclopedia.* https://en.wikipedia.org/wiki/HFS_Plus.
[6] *IOR HPC Benchmark.* http://sourceforge.net/projects/ior-sio/.
[7] *Linux Containers - LXC - Introduction.* https://linuxcontainers.org/lxc/introduction/.
[8] *Lustre.* http://lustre.org.
[9] *MDTEST. mdtest: HPC benchmark for metadata performance.* http://sourceforge.net/projects/mdtest/.
[10] *Network Common Data Form (NetCDF).* http://www.unidata.ucar.edu/software/netcdf/.
[11] *Rhea – Oak Ridge Leadership Computing Facility.* https://www.olcf.ornl.gov/computing-resources/rhea/.
[12] *RocksDB.* http://rocksdb.org.
[13] *SciDB.* http://www.paradigm4.com/.
[14] *Solr - Apache Lucene.* http://lucene.apache.org/solr/.
[15] *Spallation Neutron Source | Neutron Science at ORNL.* https://neutrons.ornl.gov/sns.
[16] *SQLite Home Page.* http://www.sqlite.org.
[17] *Storage for your Cloud. — Gluster.* http://www.gluster.org.
[18] *Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway | TOP500 Supercomputer Sites.* http://www.top500.org/system/178764.
[19] *The Large Hadron Collider | CERN.* http://home.cern/topics/large-hadron-collider.
[20] *The Large Synoptic Survey Telescope: Welcome.* http://www.lsst.org/.
[21] *Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x | TOP500 Supercomputer Sites.* http://www.top500.org/system/177975.
[22] *XGC - Oak Ridge Leadership Computing Facility.* https://www.olcf.ornl.gov/caar/xgc/.
[23] Nicolas Anciaux, Saliha Lallali, Iulian Sandu Popa, and Philippe Pucheral. 2015. A Scalable Search Engine for Mass Storage Smart Objects. *Proceedings of the VLDB Endowment* 8, 9 (2015).
[24] Anant Bhardwaj, Amol Deshpande, Aaron J Elmore, David Karger, Sam Madden, Aditya Parameswaran, Harihar Subramanyam, Eugene Wu, and Rebecca Zhang. 2015. Collaborative Data Analytics with DataHub. *Proceedings of the VLDB Endowment* 8, 12 (2015).
[25] Donald D Chamberlin, JN Gray, and Irving L Traiger. 1975. Views, Authorization, and Locking in a Relational Data Base System. In *Proceedings of the May 19-22, 1975, National Computer Conference and Exposition.*
[26] James Cipar, Greg Ganger, Kimberly Keeton, Charles B Morrey III, Craig AN Soules, and Alistair Veitch. 2012. LazyBase: Trading Freshness for Performance in a Scalable Database. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12).*
[27] W Lawrence Gates. 1992. AMIP: The Atmospheric Model Intercomparison Project. *Bulletin of the American Meteorological Society* 73, 12 (1992).
[28] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03).*
[29] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole, Jr. 1991. Semantic File Systems. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP '91).*
[30] Raghul Gunasekaran, Sarp Oral, Jason Hill, Ross Miller, Feiyi Wang, and Dustin Leverman. 2015. Comparative I/O Workload Characterization of Two Leadership Class Storage Clusters. In *Proceedings of the 10th Parallel Data Storage Workshop (PDSW '15).*
[31] Dave Hitz, James Lau, and Michael A Malcolm. 1994. File System Design for an NFS File Server Appliance. In *USENIX Winter Technical Conference.*
[32] Charles Johnson, Kimberly Keeton, Charles B Morrey III, Craig AN Soules, Alistair Veitch, Stephen Bacon, Oskar Batuner, Marcelo Condotta, Hamilton Coutinho, Patrick J Doyle, and others. 2014. From Research to Practice: Experiences Engineering a Production Metadata Database for a Scale out File System. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST '14).*
[33] Youngjae Kim, Raghul Gunasekaran, Galen M Shipman, David A Dillow, Zhe Zhang, and Bradley W Settlemyer. 2010. Workload Characterization of a Leadership Class Storage Cluster. In *Proceedings of the 5th Petascale Data Storage Workshop (PDSW '10).*
[34] Andrew W. Leung, Minglong Shao, Timothy Bisson, Shankar Pasupathy, and Ethan L. Miller. 2009. Spyglass: Fast, Scalable Metadata Search for Large-scale Storage Systems. In *Proccedings of the 7th USENIX Conference on File and Storage Technologies (FAST '09).*
[35] Jay Lofstead, Ivo Jimenez, Carlos Maltzahn, Quincey Koziol, John Bent, and Eric Barton. 2016. DAOS and Friends: A Proposal for an Exascale Storage System. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16).*
[36] Udi Manber, Sun Wu, and others. 1994. GLIMPSE: A Tool to Search Through Entire File Systems. In *Usenix Winter Technical Conference.*
[37] Marshall K McKusick, William N Joy, Samuel J Leffler, and Robert S Fabry. 1984. A Fast File System for UNIX. *ACM Transactions on Computer Systems (TOCS)* 2, 3 (1984).
[38] Sarp Oral, James Simmons, Jason Hill, Dustin Leverman, Feiyi Wang, Matt Ezell, Ross Miller, Douglas Fuller, Raghul Gunasekaran, Youngjae Kim, Saurabh Gupta, Devesh Tiwari, Sudharshan S. Vazhkudai, James H. Rogers, David Dillow, Galen M. Shipman, and Arthur S. Bland. 2014. Best Practices and Lessons Learned from Deploying and Operating Large-scale Data-centric Parallel File Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14).*
[39] Swapnil Patil and Garth Gibson. 2011. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST '09).*
[40] Robert Ross and Robert Latham. 2006. PVFS: A Parallel File System. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06).*
[41] S Sarin, Mark DeWitt, and Ronni Rosenburg. 1988. *Overview of SHARD: A System for Highly Available Replicated Data.* Technical Report. Technical Report CCA-88-01, Computer Corporation of America.
[42] Frank B Schmuck and Roger L Haskin. 2002. GPFS: A Shared-Disk File System for Large Computing Clusters.. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST '02).*
[43] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST '10).*
[44] Michael Stonebraker. 1986. The Case for Shared Nothing. *Database Engineering* 9 (1986).
[45] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06).*
[46] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. 2008. Scalable Performance of the Panasas Parallel File System.. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08).*
[47] Lei Xu, Ziling Huang, Hong Jiang, Lei Tian, and David Swanson. 2014. VSFS: A Searchable Distributed File System. In *Proceedings of the 9th Parallel Data Storage Workshop (PDSW '14).*
[48] Lei Xu, Hong Jiang, Lei Tian, and Ziling Huang. 2014. Propeller: A Scalable Real-Time File-Search Service in Distributed Systems. In *Proceedings of 2014 IEEE 34th International Conference on Distributed Computing Systems (ICDCS '14).*
[49] Dongfang Zhao, Zhao Zhang, Xiaobing Zhou, Tonglin Li, Ke Wang, D. Kimpe, P. Carns, R. Ross, and I. Raicu. 2014. FusionFS: Toward Supporting Data-Intensive Scientific Applications on Extreme-Scale High-Performance Computing Systems. In *Proceedings of 2014 IEEE International Conference on BigData (BigData '14).*