An Integrated Indexing and Search Service for Distributed File Systems

Hyogi Sim, Awais Khan, Sudharshan S. Vazhkudai, Seung-Hwan Lim, Ali R. Butt, and Youngjae Kim

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 article, we present TagIt, a scalable data management service framework aimed at scientific datasets, which can be integrated into prevalent distributed file system architectures. A key feature of TagIt is a scalable, distributed metadata indexing framework, which facilitates a flexible tagging capability to support data discovery. Furthermore, 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. We have integrated TagIt into two popular distributed file systems, i.e., GlusterFS and CephFS. Our evaluation demonstrates that TagIt can expedite data search operation by up to 10× over the extant decoupled approach.

Index Terms—Distributed systems, storage management, scientific data management

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 [43]. 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 [47], Lustre [9], HDFS [49], GlusterFS [17], Ceph [54], PanFS [57], PVFS [45], 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+ [6] for Mac OS X or Google Desktop [5], 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], [4]. 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

- *H. Sim, S. S. Vazhkudai, and S.-H. Lim are with Oak Ridge National Laboratory, Oak Ridge, TN 37830. E-mail: {simh, vazhkudaiss, lims1}@ornl.gov.*
- *A. Khan and Y. Kim are with Sogang University, Seoul 04107, South Korea. E-mail: {awais, youkim}@sogang.ac.kr.*
- *A.R. Butt is with Virginia Tech, Blacksburg, VA 24061. E-mail: butta@cs.vt.edu.*

Manuscript received 13 Aug. 2019; revised 15 Apr. 2020; accepted 21 Apr. 2020. Date of publication 27 Apr. 2020; date of current version 15 May 2020. (Corresponding author: Youngjae Kim.) Recommended for acceptance by S. He. Digital Object Identifier no. 10.1109/TPDS.2020.2990656

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 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 [11] 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 percent of their individual peak throughput for 95 percent of the time [30], [35]. While this may vary across deployments, there is the possibility of using the spare cycles for additional services.

1.1 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.

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 [6], 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. Our design does not have any centralized components and performs well even in a large-scale deployment, i.e., 105 million files in 96 logical volume servers (Section 5.1.2).

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, 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. 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. Computing the decadal average for a large atmospheric measurement data collection (a 150 GB AMIP dataset with more than 130 files) suggests that TagIt's active operator can complete 10× faster than the traditional out-of-band calculation of the average.

Automatically Extracting Metadata and Indexing. To facilitate more sophisticated searches and richer metadata, TagIt also can automatically extract metadata from files and further index them using the aforementioned active operation framework. 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.

Integration of TagIt With GlusterFS and CephFS. To show the feasibility of TagIt, we have implemented TagIt into two popular file systems, GlusterFS [19] and CephFS [2]. Our experiments on both file systems demonstrate that TagIt can provide advanced data management features with a reasonable overhead, e.g., 4 percent on GlusterFS and 13 percent on CephFS, for metadata-intensive workloads.

The rest of the paper is organized as follows: Section 2 explains the overview of TagIt followed by basic design principles and implementation of TagIt in GlusterFS in Section 3. We discuss TagIt integration in CephFS in Section 4. We present the evaluation of TagIt in GlusterFS and CephFS in Section 5. After discussing related work in Section 6, we conclude with final remarks in Section 7.

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 [51], which allows us to seamlessly integrate our ideas and demonstrate its efficacy in deployable systems.

Fig. 1 presents the architecture of TagIt. Users can read and write data objects from the file system via a mount point. 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 of GlusterFS, we have integrated data management services within the volume server to manage the metadata index

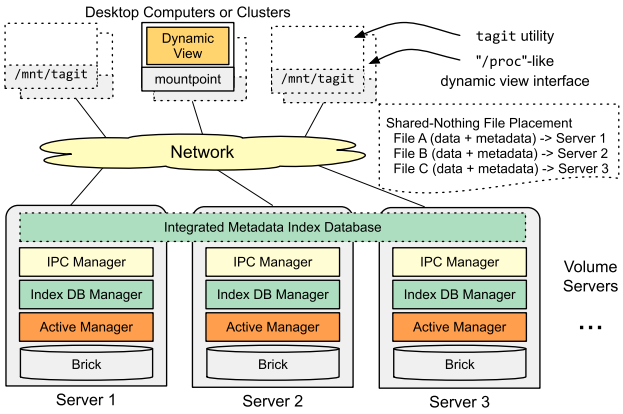


Fig. 1. Overview of TagIt architecture in GlusterFS [17].

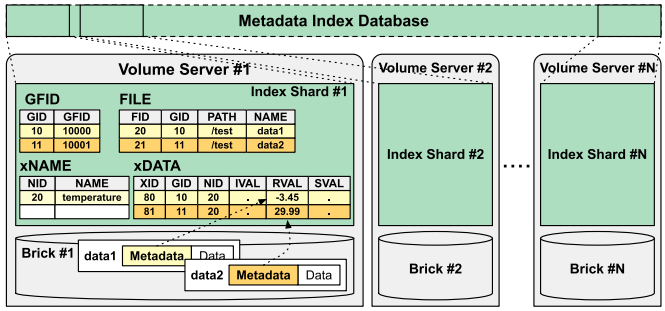


Fig. 2. Sharded metadata index database in TagIt. Each index shard is tightly coupled with the local brick.

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 TagIt in GlusterFS

In this section, we discuss the key building blocks of our approach, and how we build the metadata indexing mechanism and integrate it into 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 [41]. 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).

The relational schema is depicted in Fig. 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 $-3.45\,^{\circ}C$ and $29.99\,^{\circ}C$ 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* [46], 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 Section 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. To distribute the index shards, TagIt simply follows the file distribution algorithm of the underlying GlusterFS, i.e., each index shard is populated with the metadata of files that are locally stored in the corresponding volume server (Fig. 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, our distribution mechanism 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 [43], 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 (Section 5.1.1). 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

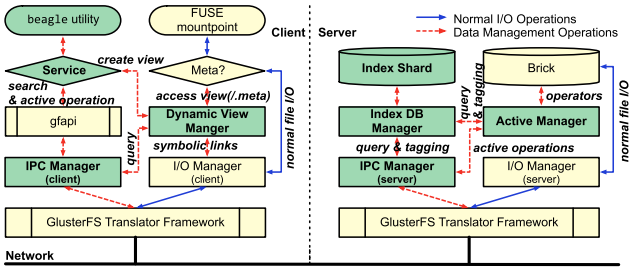


Fig. 3. Service architecture of TagIt.

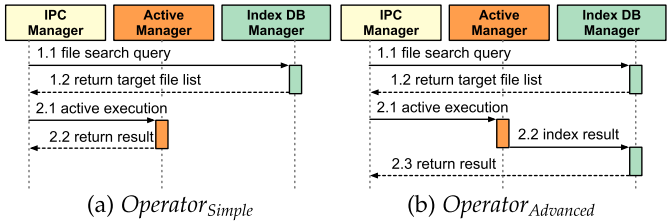


Fig. 4. Control flows of active operators inside a volume server.

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 broadcasts search queries to all distributed index shards. Despite of the communication overhead, TagIt can achieve significantly improved performance particularly for complicated file search queries, even in a large-scale cluster, i.e., 96 volume servers (Section 5.1.2).

Also, the distributed query processing in TagIt can impact the performance of foreground I/O operations. However, in a practical scientific workflow, i.e., searching for a dataset and then applying an operation on them, TagIt not only outperforms a conventional method (without using TagIt) but also impacts less on other foreground I/O operations (Section 5.1.3).

3.5 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)`. Fig. 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.

3.6 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 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 64 KB 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 Fig. 4a, and it is referred as $Operator_{simple}$. Particularly for data-intensive tasks, using the active operators can effectively decrease the application runtime by reducing data transfers between the client and file servers, and also by exploiting the parallelism from multiple volume servers.

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 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 Fig. 4b 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 Linu Container [8] 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 4 GB memory to the container from a 12 core, 64 GB volume server in our testbed (Table 1). Further exploration for building a secure environment is beyond the scope of this work.

TABLE 1
Testbed Specification

	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	240 GB SSD, 1 TB HDD	N/A

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. TagIt provides a dynamic view, a special file that associated to a search query. This concept is similar to *views* in relational databases [25] and further elaborated in our previous work [50].

4 TagIt in CephFS

In this section, we explain how TagIt has been integrated into CephFS [2]. Although both GlusterFS and CephFS share the *shared-nothing* design philosophy, the architectures

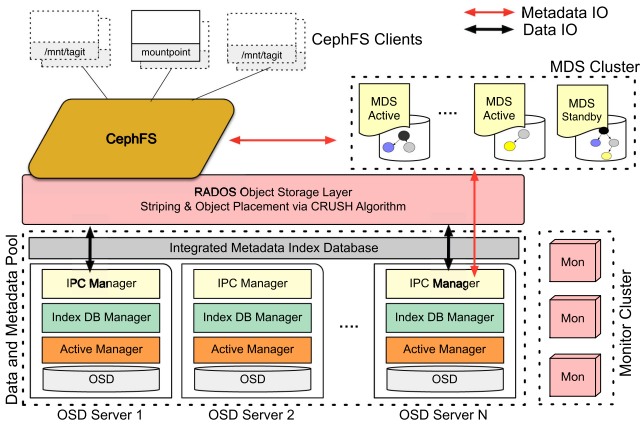


Fig. 5. Overview of TagIt architecture when integrated in CephFS.

of two file systems fundamentally differ. Therefore, we first explain the architecture of CephFS (Section 4.1) and then discuss our integration of TagIt in CephFS (Section 4.3).

4.1 CephFS Architecture

Ceph is a distributed storage system that offers versatile storage interfaces such as object, block, and file [2]. Particularly, the scalable design principles of Ceph such as clean storage abstraction, decoupling data and control paths, and deterministic object placement allow to facilitate high-performance file systems at scale [55], [56]. CephFS is built upon the Ceph object storage, RADOS [56], by layering the file system interface and semantics. CephFS follows a shared-nothing architecture and thus effectively avoids common problems in distributed storage systems, including the single point of failure and the centralized hotspot. Internally, a CephFS cluster consists of three major components, i.e., Object Storage Daemon (OSD), Monitor (Mon), and Metadata Server (MDS), as shown in Fig. 5. Next, we briefly explain each of these three components.

4.1.1 Object Storage Daemon (OSD)

Ceph OSD is primarily responsible for storing and managing object data and metadata. In addition, OSD also performs internal cluster operations such as replication, backfilling (for failure recovery [55]), and heartbeat exchanges between peers. An OSD can be associated with a single disk or partition, and thus a single physical server can host multiple OSD daemons. Ceph defines logical OSD pools, each of which consists of a set of OSDs. A single logical pool is further grouped into multiple placement groups (PGs). A number of operations in Ceph, including object placement and replication, are performed at the granularity of a PG. For instance, the CRUSH algorithm [34], [55] deterministically computes a target PG for a given object and its attributes. The RADOS layer provides uniform access method to the object storage layer, as depicted in Fig. 5.

4.1.2 Metadata Server (MDS)

As mentioned above, CephFS layers the file system interface and semantics atop the object storage, RADOS. Internally, CephFS defines a MDS cluster (Fig. 5), a dedicated set of servers in the cluster for serving file system metadata operations. The MDS cluster does not store the file system metadata locally but persist them using the RADOS object storage layer. A set of OSDs can form a metadata pool for handling metadata requests from the MDS cluster. For handling operations of a file, a CephFS client first needs to acquire a file handler and capability of the file from the MSD cluster [48]. The file handler contains the file attributes, e.g., ownership and striping information, while the capability is a public key that allows the client to directly access to the target OSDs. After acquiring the file handle and capability, the client can directly access the target OSDs for subsequently manipulating the file data. To reduce the metadata workload on OSD servers, the MDS cluster keeps the directory map in the collective memory space and acts as a distributed metadata cache. CephFS also introduces a dynamic subtree partitioning algorithm [48], for dynamically balancing the workload across the MDS cluster.

4.1.3 Monitor

A set of Monitor daemons manage the topology of the CephFS cluster. For instance, a CephFS client first asks the Monitor daemon of the latest cluster map, which describes the cluster topology and authorization information, for sending I/O requests to appropriate service daemons. Specifically, the cluster map consists of five maps, i.e., Monitor, OSD, MDS, PG, and CRUSH maps [54].

4.2 CephFS and GlusterFS: Architectural Analogy

Here, we summarize key design differences between GlusterFS and CephFS. Particularly, we focus on architectural aspects that directly affect our design of TagIt in both file systems. Figs. 1 and 5 depicts the high-level architecture of GlusterFS and CephFS, respectively, coupled with TagIt.

Data and Metadata Distribution. Although, both file systems employ a hash-based placement algorithm for evenly distributing file system objects, e.g., metadata records, data blocks, etc., across the file system cluster, actual object management is fundamentally different from each other. Specifically, GlusterFS determines a target server of a file according to the hash value of the file name, and then stores both data and metadata of the file to the target server. In addition, the directory tree is replicated in all backend file systems in the cluster. Therefore, all operations to a file can be isolated to a single server in GlusterFS. In contrast, CephFS defines a data pool for storing data objects and separately a metadata pool for storing metadata objects. However, both of data and metadata pools can be configured to be an identical set of servers. Consequently, depending on the cluster configuration, a data object and its corresponding metadata object in CephFS can be stored in a single server or two different servers. Therefore, integration of TagIt on CephFS differs from integration of TagIt in GlusterFS due to disjoint and loosely coupled placement of data and metadata.

Metadata Extraction I/O Path. For initiating operations to a file, a CephFS client first need to acquire the file layout information from a MDS. This I/O path is intrinsically different from I/O paths in GlusterFS, which does not have any dedicated metadata servers. Therefore, in CephFS, TagIt collects standard file system metadata, i.e., `stat(2)` attributes, from OSD servers in the metadata pool, while it extracts additional metadata from the file content from OSD servers in the data pool. In contrast, TagIt in GlusterFS

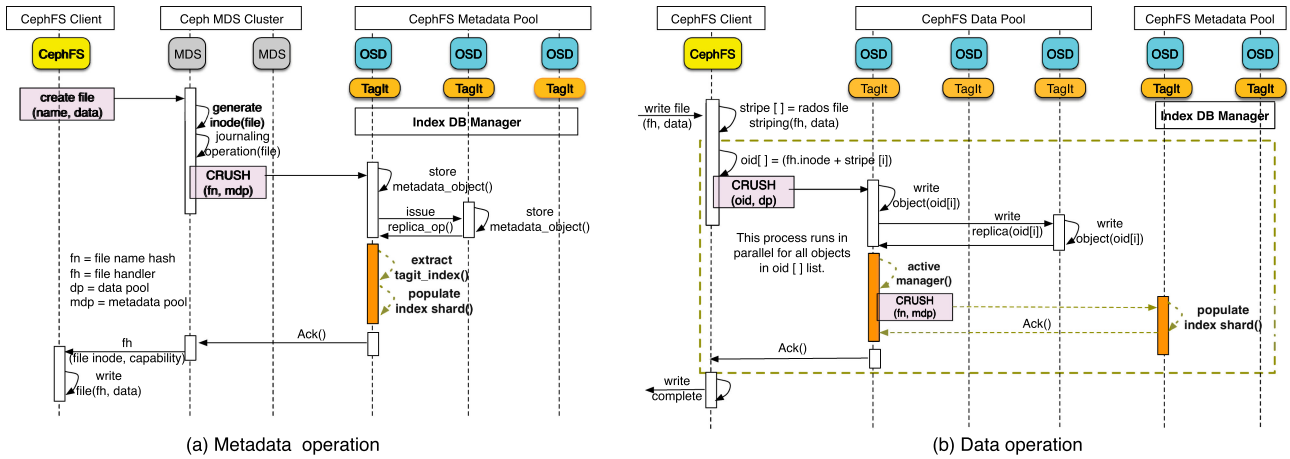


Fig. 6. Updated (a) metadata operation path and (b) data operation path in CephFS with the TagIt integration. The TagIt operations are shown in bold fonts.

collects and extracts all metadata of a file from a single volume server.

For integrating TagIt into CephFS, aforementioned architectural dissimilarities between CephFS and GlusterFS impose unavoidable changes to our original design of TagIt in GlusterFS. In the following subsection, we explain how TagIt is attuned to the CephFS architecture.

4.3 TagIt Integration in CephFS

Here, we elaborate how we have integrated TagIt into the CephFS architecture.

4.3.1 Metadata Index Distribution

Fig. 5 depicts the architecture of TagIt in CephFS. As discussed above, the CephFS cluster largely grouped into three internal clusters, i.e., OSD, MDS, and Monitor clusters. The architecture of CephFS differs from GlusterFS primarily in handling the file system metadata. In GlusterFS, a single volume server stores both data and metadata of a file (Section 2), and a client can perform both data and metadata operations of a file by exploiting a single volume server. In contrast, CephFS keeps a separate MDS cluster for handling metadata operations. Moreover, CephFS stores file data and metadata using separate OSD pools, meaning that it is not likely that a single server stores both data and metadata of a file. This dissimilarity in handling the file metadata imposes a major challenge for integrating the TagIt framework into CephFS.

In our design, we host the index shards on the OSD servers in the metadata pool. Specifically, each OSD in the metadata pool hosts its own index database shard, which is then populated by the file metadata that the corresponding OSD stores. This effectively co-locates the original file metadata and the associated TagIt index records, similarly to TagIt in GlusterFS (Section 3.2), and provides the following advantages compared to alternatively placing the index shards on the MDS cluster. First, the number of OSD servers tend to far exceed the number of MDS servers in a prudently configured CephFS cluster, i.e., the recommended number of MDS servers is only one tenth of the number of OSDs in a cluster [3], [53]. Therefore, TagIt can reduce the performance impact on each server and also exploit more aggregated computational power for active operations. Second, we can distribute the index shard records by simply following the native metadata distribution mechanism in CephFS, i.e., the CRUSH algorithm. Third, in case of a failure, the recovery process becomes simpler by hosting the TagIt records in the same OSD that stores the corresponding CephFS file metadata.

4.3.2 Index Database Update

Index Update. Fig. 6 shows the extended I/O paths of CephFS in (a) file metadata and (b) data operations with the TagIt integration. Specifically, the operations in bold fonts highlight the additional steps that TagIt introduces.

For a file creation (Fig. 6a), a CephFS client first sends a file creation request to an active MDS server. Upon receiving the request, the MDS server first determines the file inode number and the target OSD server within the metadata pool (via the CRUSH algorithm) and then journals the metadata operation to ensure the file system consistency on failures. Later, the MDS server asynchronously flushes the journal data to OSD servers via the RADOS interface. An important characteristic of CRUSH algorithm is that it is fully distributed, implying that any client, MDS or OSD can independently determine the location of any object in the cluster based on the target object name. On the target OSD server, the metadata object is stored in the form of a key-value pair, where the key is file name hash and the value is file layout attributes [54]. In addition, the metadata object can be replicated in multiple OSDs based on the configurable replication factor [56]. The TagIt service is triggered when the OSD server handles the replication. Specifically, TagIt reads the target metadata object and populate the index shard synchronously with file system metadata, as shown in Fig. 6a. After index shard is populated, an acknowledgement is sent to the MDS server, which then responds to the client with a file handler and capability [37]. The MDS server also caches the metadata entry in memory for handling future operations.

Fig. 6b depicts the flow of data operations, after the file metadata has been created and populated in the TagIt index shard. First, the CephFS client passes the file data to the underlying RADOS layer with the file handler and metadata information. Then, RADOS stripes the file into smaller fixed unit objects, or stripes (4 MB by default), and generates an object identifier (object id) for each stripe. After this, the CephFS uses CRUSH [55] algorithm to distribute the

stripes across the data pool OSD servers in parallel [56]. Similarly to the metadata objects, these stripe objects are also replicated based on the replication policy. At this point, TagIt processes the file data for user-defined tags if necessary, e.g., metadata extraction. After processing the data, TagIt may need to send the metadata to a different OSD, because CephFS differentiates the data pool and metadata pool. TagIt acquires the target OSD by using the CRUSH algorithm (with the target filename), as shown in Fig. 6b. Upon receiving the data, the target OSD properly updates the file attributes in the index shard and sends an acknowledgment. These steps are repeated until all stripes are exhausted, at which the CephFS client receives an acknowledgement message indicating a success. Note that TagIt operation paths reside strictly within the original file operation paths of CephFS, which obviates the need for implementing its own transactions to assure the consistency across multiple index shards. In addition, TagIt does not disrupt any distributed transaction semantics that are required by file operations in CephFS.

Consistency. With TagIt-Async, users may experience a brief inconsistency between the file system and index database. Similarly to TagIt in GlusterFS (Section 3.3), TagIt-Sync or the `tagit-sync` utility can be used appropriately to guarantee the stronger consistency. Furthermore, TagIt does not affect the consistency of file operations in CephFS.

Durability. For ensuring the file system consistency across system failures, CephFS maintains the file system journal on MDS. When CephFS performs the recovery process by replaying the journal, some metadata objects may be appended or removed on metadata pool OSDs. To keep the index database consistent with the recovered file system status, TagIt deliberately suspend its recovery process until CephFS completes the recovery. Once the file system is fully recovered, the TagIt instance in each OSD server examines the changes in the metadata objects and updates the index database accordingly. Therefore, as with TagIt in GlusterFS (Section 3.3), the recovery process of TagIt in CephFS is locally performed within individual OSDs.

4.3.3 Distributed Query Processing

To facilitate scientific discovery processes, TagIt supports sophisticated file search queries based on file attributes and user taggings. Similarly to TagIt in GlusterFS, the query processing of TagIt in CephFS broadcasts a search query to all index shards in the cluster. However, the different cluster architecture in CephFS results in different performance implications in processing the file search queries.

As we discussed in Section 4.1, CephFS allows to assign an exclusive set of OSDs, i.e., a metadata pool, for storing the file system metadata, and effectively separates the metadata and data paths within the cluster [54]. Since TagIt populates the index shards only in the metadata pool, the number of TagIt index shards in CephFS becomes substantially smaller than the number of index shards in GlusterFS. Consequently, the query broadcasting overhead also lessens in CephFS. Furthermore, due to the strict separation of data and metadata paths in CephFS, the distributed query processing of TagIt in CephFS does not directly affect the foreground data I/O operations.

Other TagIt services such as active operators are executed on data pool OSD servers. The primary architecture is similar to our design in GlusterFS (Section 3.4), and we do not elaborate here.

5 Evaluation

In this section, we evaluate the performance of TagIt in GlusterFS (Section 5.1) and CephFS (Section 5.2).

5.1 Evaluation of TagIt in GlusterFS

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. 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 (Section 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` [10] and `IOR` [7] 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 percent confidence interval.

5.1.1 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. Fig. 7 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.

Fig. 7a, 7b, 7c, and 7d show the results with the SSD volume configuration. In file operations (Fig. 7a and 7b), 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 4 percent 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 (Fig. 7c and 7d), 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 (Section 2). Fig. 7e, 7f, 7g, and 7h show the results with the HDD volume configuration. Not surprisingly, we have similar observations as in Fig. 7a, 7b, 7c, and

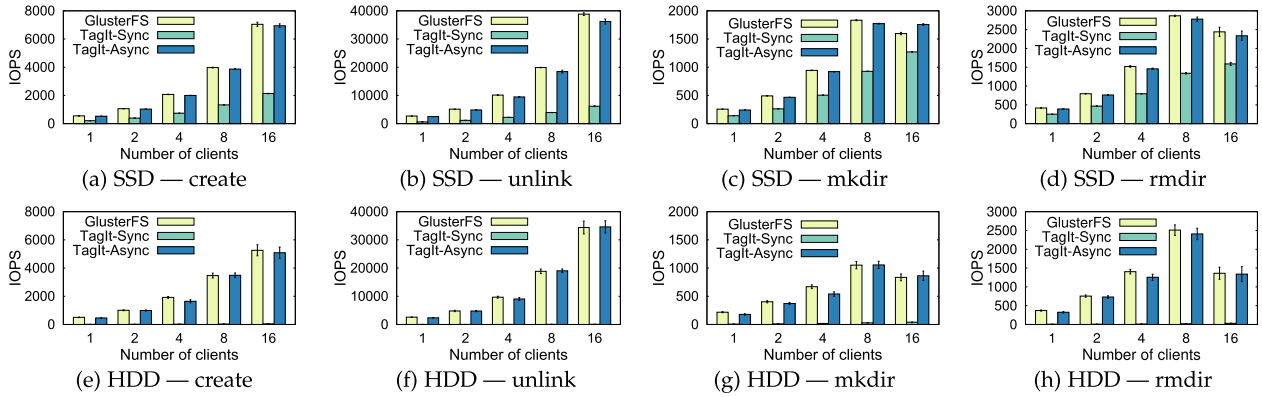


Fig. 7. Performance overhead of metadata indexing in the file system. `mdtest` [10] benchmark was used to generate metadata-intensive workloads. We used two different storage volume configurations, with SSDs ((a)–(d)) and with HDDs ((e)–(h)), to observe the performance impact of storage device characteristics.

7d, 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 Fig. 8a, 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 (Section 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 (Section 3.3). To investigate the delay, we measured database update latencies of the first 10,000 create requests. Fig. 8b 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. Fig. 9 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.

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, 10 Gbit/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). Fig. 10 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

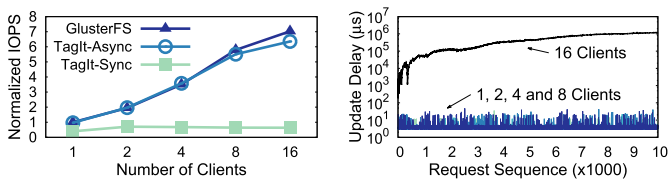


Fig. 8. Experiments with an overloaded server. (a) shows the normalized throughput, and (b) depicts queueing delays of database update requests.

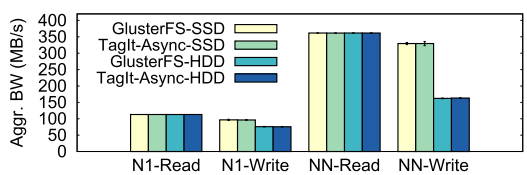


Fig. 9. Performance comparison of GlusterFS and TagIt-Async for parallel I/O workloads. `IOR` benchmark [7] was used to generate N1 and NN workdloads.

Fig. 10. Metadata indexing overhead of TagIt for a large deployment. *F-* and *D-* denote the file and directory operations, respectively.

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 percent in all cases, except for the file remove operation (F-remove) where the overhead is around 10 percent. Since file remove (unlink) is the fastest metadata operation in GlusterFS (Fig. 7), 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.

5.1.2 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 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 [43], 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 [38]. 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 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 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 Fig. 11. In Fig. 11, 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.

TABLE 2
Various File Search Queries to Measure the Query Performance

	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

Attribute column shows metadata required to answer the query, while table column shows database tables that hold the metadata columns.

TABLE 3
Query Performance Under TagIt Versus the Crawling Approach With 16 MySQL Servers

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

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.

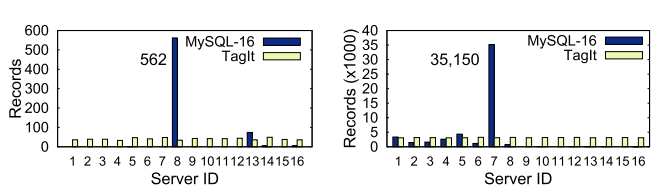
Such a skewed distribution of records not only negates the benefit of the parallel query processing, but also significantly slows down the overall processing time. Note that a single query processing internally involves communication with all 16 database servers due to the nature of the sharded database 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 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 (Section 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 [12]. 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

TABLE 4
Database Size and Update Time Under TagIt Versus the Crawling Approach With 16 MySQL Servers

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



(a) Q4 — 647 records (b) Q5 — 50,552 records

Fig. 11. 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.

TABLE 5
Coefficients of Linear Runtime Functions With the Number of Clients as an Explanatory Variable

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

In all cases, R^2 values are greater than 99 percent.

this setup. Fig. 12 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).

5.1.3 Science Discovery Services

To study the effectiveness of active operators, we used the query of computing the decadal average temperature of the AMIP [27] datasets, composed of 132 1.2 GB netCDF files (total 150 GB) (Section 3.6). We wrote a dedicated program (operator) using the netCDF library, which that 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.

Evaluation of Operator$_{simple}$ Fig. 13a shows the results without any foreground I/O. For *Offline*, the run time decreases as we increase the parallelism up to 4 clients. With 8 clients, however, the effect of parallelism almost disappears due to 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 near-data processing, minimizing data movement between the file servers and the client. Moreover, due to the shared nothing architecture of TagIt, there exists little I/O contention between the operators running on the different servers. Fig. 13b shows the results when either

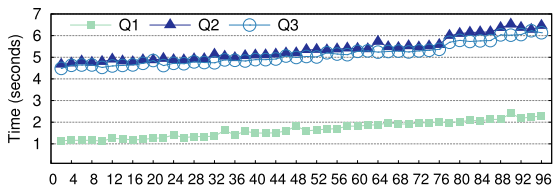
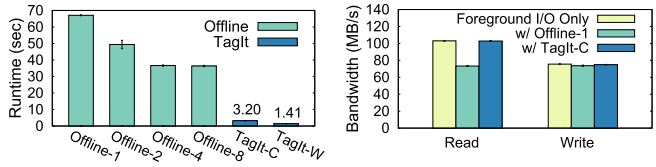


Fig. 12. Query performance scaling under increasing volume servers and 105 million files.



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

Fig. 13. Performance impact of active operators in TagIt. (a) Performance under Offline versus TagIt. (b) Impact on foreground I/O operations. TagIt-C and TagIt-W show the cold and warm volume server cache case, respectively.

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 percent. 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 locally buffered before reaching the servers and thus not directly affected by the server-side contentions.

Evaluation of Operator$_{advanced}$ We have also evaluated the performance impact of Operator$_{\text{advanced}}$. Our experiment result indicates that Operator$_{\text{advanced}}$ perform 10 percent faster than Operator$_{\text{simple}}$ to calculate statistical summaries of the same AMIP dataset [50].

5.2 Evaluation of TagIt in CephFS

Implementation. TagIt has been implemented atop Ceph 10.2.3. Specifically, we have extended the Ceph OSD service to manage a dedicated index database shard using SQLite [16]. For the command-line utilities, we have extended the existing Ceph command line interface. We compare the performance of TagIt-Sync and TagIt-Async against the baseline CephFS without any modification.

Testbed. We configured a Ceph storage cluster consisting of eight OSD servers (OSSs), three Monitors, four Metadata servers and two Ceph client nodes. All nodes are connected via a 10 Gbps network. Each machine is equipped with Intel E5-2670v4@2.40 GHz (10 Cores) and 32 GB DRAM, running CentOS 7.3. In addition, each OSS also has two 256 GB SSDs. Each SSD is formatted as an XFS volume and dedicated to a single OSD daemon. Therefore, each OSS spawns two OSD daemons, and the Ceph cluster runs 16 OSD daemons in total. To minimize the performance variance, we disable the MDS logging and also discard the client side inode, dentry and page caches before each run.

For performance comparison, we use the `mdtest` benchmark [10], which generates metadata operations in parallel. All experiments were repeated six times, and the performance variance was small between runs, i.e., less than 2 percent. Therefore, we simply report an average.

5.2.1 Metadata-Intensive Workloads

Fig. 14 shows the performance comparison between TagIt-Sync, TagIt-Async and CephFS for metadata-intensive workloads, including file operations, i.e., (a) create and (b) unlink, and directory operations, i.e., (c) mkdir and (d) rmdir. To mimic a congested environment, we increase the number of client threads in each client node up to 8, i.e., total 16 clients.

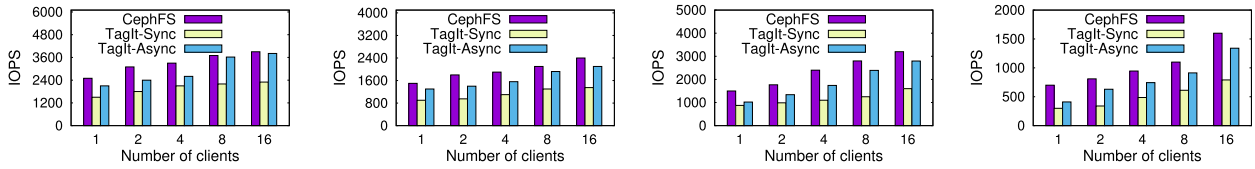


Fig. 14. Performance overhead of metadata indexing in the CephFS. We used the `mdtest` [10] benchmark for generating the metadata-intensive workloads.

For all four operations, we observe that both TagIt and CephFS scale linearly with respect to the number of clients. On average, the throughput of TagIt-Sync is 40 percent lower than the throughput of CephFS, due to its additional operations for updating the index database shard in a synchronous manner. Specifically, SQLite frequently performs file sync operations, which leads to a significant delay. However, we note that the performance degradation of TagIt-Sync in CephFS is relatively lower, compared to our previous observation for TagIt-Sync in GlusterFS (Fig. 7). This is because of the primary design of the Ceph file system, i.e., i) decoupled data and metadata management, ii) file and directory metadata in CephFS is very small, consisting entirely of directory entries (file names) and inodes only, and iii) backend key-value database for storing metadata objects [54]. For TagIt-Async, the average performance reduction across the four metadata operations is less than 13 percent, a reasonable trade-off for providing advanced data management features.

5.2.2 Impact of Varying the Metadata Pool Size

The size of the metadata pool in CephFS, i.e., the number of OSDs in the metadata pool, is configurable. This allows us to infer the performance impact of TagIt in other file systems with a different number of metadata servers. For instance, Lustre [9] runs a single metadata server, while GlusterFS utilizes all available servers for storing metadata (Section 2). Therefore, in this experiment, we increase the size of the metadata pool in our CephFS cluster up to 16 and measure the performance overhead of TagIt. Specifically, we spawn 16 client threads, each of which creates 5,000 files in its own directory.

Fig. 15 shows the file create performance of CephFS, TagIt-Sync, and TagIt-Async under the different metadata pool sizes. We first observe that the performance of the metadata operation severely diminishes in all systems, when the number of OSDs becomes less than four. On average, TagIt-Sync exhibits a performance overhead of 70 percent, whereas the performance reduction of TagIt-Async is 16 percent. Particularly, the overhead of TagIt tends to become higher when OSDs are heavily congested, e.g., the performance overhead of TagIt-Sync and TagIt-Async is 77 and 27 percent, respectively, when only a single OSD is allocated. This performance degradation is also attributed to the design of the placement group (PG) in CephFS. Specifically, the smaller metadata pool leads to a less number of PGs for servicing the incoming requests. Since CephFS uses a PG-based locking, all the requests landing to a single PG exhibit a significant delay from the PG lock contention [42]. Moreover, TagIt-Sync further increases the latency for populating the associated entries synchronously in the TagIt index shard. In contrast, CephFS and TagIt-Async exhibit stable performance and scalability with the increasing number of OSDs, as our previous observation in Fig. 14.

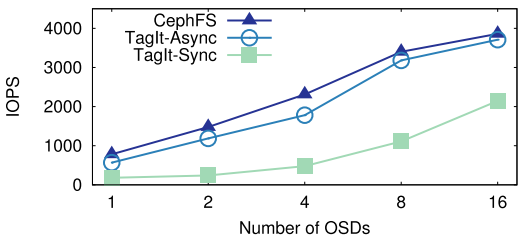


Fig. 15. Comparative performance analysis of CephFS and TagIt while varying the number of OSDs in metadata pool.

Overall, our experiments with CephFS demonstrate that the primary concept of TagIt, i.e., integrating the data management services into a file system, is feasible and practical for diverse distributed file systems.

6 Related Work

Managing metadata in a large-scale file system has been the focus of many works. GIGA+ [44] is a directory service that can be stacked on any parallel file systems. FusionFS [61] and SoMeta [52] employ a distributed key-value store for a scalable metadata management. Recently, DAOS [39] 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 Section 3.1.

File system searchability has mostly been achieved by using external applications in a post hoc fashion [5], [40]. 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 [38] 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 [58] 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 [59]. The integrated design of TagIt precludes such extra servers and custom distributions. HP StoreAll ExpressQuery [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 (Section 3.4). 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. DART [60] is a distributed data structure that is tailored to support affix-based keyword search operations in HPC systems. However, its inability to support numerical range queries may diminish its practicality in scientific computing. In contrast, TagIt supports complex search operations including the numerical range queries and string-based queries.

In scientific computing, a number of custom solutions, i.e., external to the file system, have been proposed to provide data management services. 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. EMPRESS 2.0 [36] stores custom metadata of a simulation outcome using a relational database to facilitate a post-processing. SciSpace [33] offers search and discovery services based on a collaborative scientific namespace framework. However, such designs require using a custom interface instead of a file system and may create an 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 two popular distributed file systems, i.e., GlusterFS and CephFS, shows that TagIt is viable and outperforms an external data management approach, without the need for deploying any additional resources.

Acknowledgments

This research was supported in part by the U.S. DOE's Scientific data management program, by US National Science Foundation through Grants CNS-1615411, CNS-1405697 and CNS-1565314, and by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (Ministry of Science and ICT) under Grant 2018R1A1A1A05079398. 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).

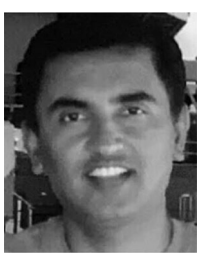
References

[1] ARM Climate Research Facility, 2020. [Online]. Available: http://www.arm.gov/.
[2] Ceph Home Page - Ceph, 2020. [Online]. Available: https://ceph.com/
[3] Discover CephFS, 2020, Accessed: Aug. 07, 2019. [Online]. Available: https://www.suse.com/media/report/discover_cephfs.PDF
[4] ESGF, Earth System Grid Federation, 2017. [Online]. Available: http://esg.ccs.ornl.gov
[5] Google Desktop, 2017, [Online]. Available: http://desktop.google.com
[6] HFS Plus - Wikipedia, the free encyclopedia, 2020. [Online]. Available: https://en.wikipedia.org/wiki/HFS_Plus
[7] IOR HPC Benchmark, 2020. [Online]. Available: http://sourceforge.net/projects/ior-sio/
[8] Linux Containers - LXC - Introduction, 2020. [Online]. Available: https://linuxcontainers.org/lxc/introduction/
[9] Lustre, 2020. [Online]. Available: http://lustre.org
[10] MDTEST. mdtest: HPC benchmark for metadata performance, 2020. [Online]. Available: http://sourceforge.net/projects/mdtest/
[11] Network Common Data Form (NetCDF), 2020. [Online]. Available: http://www.unidata.ucar.edu/software/netcdf/
[12] Rhea – Oak Ridge Leadership Computing Facility, 2020. [Online]. Available: https://www.olcf.ornl.gov/computing-resources/rhea/
[13] SciDB, 2020. [Online]. Available: http://www.paradigm4.com/
[14] Solr - Apache Lucene, 2020. [Online]. Available: http://lucene.apache.org/solr/
[15] Spallation Neutron Source | Neutron Science at ORNL, 2020. [Online]. Available: https://neutrons.ornl.gov/sns
[16] SQLite Home Page, 2020. [Online]. Available: http://www.sqlite.org
[17] Storage for your Cloud. — Gluster, 2020. [Online]. Available: http://www.gluster.org
[18] Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway | TOP500 Supercomputer Sites, 2020. [Online]. Available: http://www.top500.org/system/178764
[19] The Large Hadron Collider | CERN, 2020. [Online]. Available: http://home.cern/topics/large-hadron-collider
[20] The Large Synoptic Survey Telescope: Welcome, 2020. [Online]. Available: http://www.lsst.org/
[21] Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x | TOP500 Supercomputer Sites. [Online]. Available: http://www.top500.org/system/177975
[22] XGC - Oak Ridge Leadership Computing Facility, 2020. [Online]. Available: https://www.olcf.ornl.gov/caar/xgc/
[23] N. Anciaux, S. Lallali, I. S. Popa, and P. Pucheral, "A scalable search engine for mass storage smart objects," *Proc. VLDB Endowment*, vol. 8, no. 9, pp. 910–921, 2015.
[24] A. Bhardwaj *et al.*, "Collaborative data analytics with DataHub," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1916–1919, 2015.
[25] D. D. Chamberlin, J. Gray, and I. L. Traiger, "Views, authorization, and locking in a relational data base system," in *Proc. May 19–22, 1975 Nat. Comput. Conf. Expo.*, 1975, pp. 425–430.
[26] J. Cipar, G. Ganger, K. Keeton, C. B. Morrey III , C. A. Soules, and A. Veitch, "LazyBase: Trading freshness for performance in a scalable database," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 169–182.
[27] W. L. Gates, "AMIP: The atmospheric model intercomparison project," *Bull. Amer. Meteorological Soc.*, vol. 73, no. 12, pp. 1962–1970, 1992.
[28] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proc. 19th ACM Symp. Operating Syst. Princ.*, 2003, pp. 29–43.
[29] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole, Jr., "Semantic file systems," in *Proc. 13th ACM Symp. Operating Syst. Princ.*, 1991, pp. 16–25.
[30] R. Gunasekaran, S. Oral, J. Hill, R. Miller, F. Wang, and D. Leverman, "Comparative I/O workload characterization of two leadership class storage clusters," in *Proc. 10th Parallel Data Storage Workshop*, 2015, pp. 31–36.
[31] D. Hitz, J. Lau, and M. A. Malcolm, "File system design for an NFS file server appliance," in *Proc. USENIX Winter Tech. Conf.*, 1994, Art. no. 19.

[32] C. Johnson *et al.*, "From research to practice: Experiences engineering a production metadata database for a scale out file system," in *Proc. 12th USENIX Conf. File Storage Technol.*, 2014, pp. 191–198.
[33] A. Khan, T. Kim, H. Byun, and Y. Kim, "SciSpace: A scientific collaboration workspace for geo-distributed HPC data centers," *Future Gener. Comput. Syst.*, vol. 101, pp. 398–409, 2019.
[34] A. Khan, C. Lee, P. Hamandawana, S. Park, and K. Youngjae, "A robust fault-tolerant and scalable cluster-wide deduplication for shared-nothing storage systems," in *Proc. IEEE 26th Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2018, pp. 87–93.
[35] Y. Kim, R. Gunasekaran, G. M. Shipman, D. A. Dillow, Z. Zhang, and B. W. Settlemyer, "Workload characterization of a leadership class storage cluster," in *Proc. 5th Petascale Data Storage Workshop*, 2010, pp. 1–5.
[36] M. Lawson and J. Lofstead, "Using a robust metadata management system to accelerate scientific discovery at extreme scales," in *Proc. IEEE/ACM 3rd Int. Workshop Parallel Data Storage Data Intensive Scalable Comput. Syst.*, 2018, pp. 13–23.
[37] A. Leung and E. L. Miller, "Scalable security for large, high performance storage systems," in *Proc. 2nd ACM Workshop Storage Secur. Survivability*, 2006, pp. 29–40.
[38] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller, "Spyglass: Fast, scalable metadata search for large-scale storage systems," in *Proc. 7th USENIX Conf. File Storage Technol.*, 2009, pp. 153–166.
[39] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton, "DAOS and friends: A proposal for an exascale storage system," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, Art. no. 50.
[40] U. Manber *et al.*, "GLIMPSE: A tool to search through entire file systems," in *Proc. Usenix Winter Tech. Conf.*, 1994, Art. no. 4.
[41] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A fast file system for UNIX," *ACM Trans. Comput. Syst.*, vol. 2, no. 3, pp. 181–197, 1984.
[42] M. Oh, J. Eom, J. Yoon, J. Y. Yun, S. Kim, and H. Y. Yeom, "Performance optimization for all flash scale-out storage," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2016, pp. 316–325.
[43] S. Oral *et al.*, "Best practices and lessons learned from deploying and operating large-scale data-centric parallel file systems," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 217–228.
[44] S. Patil and G. Gibson, "Scale and concurrency of GIGA+: File system directories with millions of files," in *Proc. 9th USENIX Conf. File Stroage Technol.*, 2011, Art. no. 13.
[45] R. Ross and R. Latham, "PVFS: A parallel file system," in *Proc. ACM/IEEE Conf. Supercomput.*, 2006, pp. 1135–1142.
[46] S. Sarin, M. DeWitt, and R. Rosenburg, "Overview of SHARD: A system for highly available replicated data," Comput. Corporation America, Cambridge, MA, USA, Tech. Rep. CCA-88–01, 1988.
[47] F. B. Schmuck and R. L. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proc. 1st USENIX Conf. File Storage Technol.*, 2002, pp. 19–es.
[48] M. A. Sevilla *et al.*, "Mantle: A programmable metadata load balancer for the ceph file system," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2015, pp. 1–12.
[49] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–10.
[50] 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 *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2017, Art. no. 5.
[51] M. Stonebraker, "The case for shared nothing," *Database Eng.*, vol. 9, pp. 1–12, 1986.
[52] H. Tang, S. Byna, B. Dong, J. Liu, and Q. Koziol, "SoMeta: Scalable object-centric metadata management for high performance computing," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2017, pp. 359–369.
[53] A. W. Leung and E. Miller, "Scalable security for large, high performance storage systems," in *Proc. 2nd ACM Workshop Storage Secur. Survivability*, 2006, pp. 29–40.
[54] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proc. 7th Symp. Operating Syst. Des. Implementation*, 2006, pp. 307–320.
[55] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "CRUSH: Controlled, scalable, decentralized placement of replicated data," in *Proc. ACM/IEEE Conf. Supercomput.*, 2006, pp. 122–es.
[56] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn, "RADOS: A scalable, reliable storage service for petabyte-scale storage clusters," in *Proc. 2nd Int. Workshop Petascale Data Storage: Held Conjunction Supercomput.*, 2007, pp. 35–44.
[57] B. Welch *et al.*, "Scalable performance of the Panasas parallel file system," in *Proc. 6th USENIX Conf. File Storage Technol.*, 2008, pp. 1–17.
[58] L. Xu, Z. Huang, H. Jiang, L. Tian, and D. Swanson, "VSFS: A searchable distributed file system," in *Proc. 9th Parallel Data Storage Workshop*, 2014, pp. 25–30.
[59] L. Xu, H. Jiang, L. Tian, and Z. Huang, "Propeller: A scalable real-time file-search service in distributed systems," in *Proc. IEEE 34th Int. Conf. Distrib. Comput. Syst.*, 2014, pp. 378–388.
[60] W. Zhang, H. Tang, S. Byna, and Y. Chen, "DART: Distributed adaptive radix tree for efficient affix-based keyword search on HPC systems," in *Proc. 27th Int. Conf. Parallel Archit. Compilation Techn.*, 2018, Art. no. 24.
[61] D. Zhao *et al.*, "FusionFS: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems," in *Proc. IEEE Int. Conf. Big Data*, 2014, pp. 61–70.

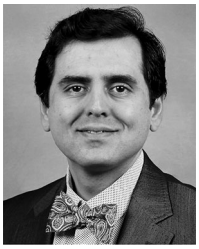
Hyogi Sim received the BS degree in civil engineering and MS degree in computer engineering from Hanyang University, Seoul, South Korea, and the MS degree in computer science from Virginia Tech, Blacksburg, Virginia, in 2014, and is currently working toward the PhD degree at Virginia Tech, Blacksburg, Virginia. He joined Oak Ridge National Laboratory in 2015, as a post-masters associate. During this appointment, he conducted research and development on active storage systems and scientific data management for HPC systems. He is currently an HPC systems engineer with Oak Ridge National Laboratory. His primary role is to design and develop a checkpoint-restart storage system for the exascale computing project. His areas of interest include storage systems and distributed systems.

Awais Khan received the BS degree in bioinformatics from Mohammad Ali Jinnah University, Islamabad, Pakistan. He is an MS leading to PhD integrated program student in Sogang University, Seoul, South Korea. He worked for one of leading software companies as a software engineer from 2012 to 2015. Currently, he is a member with Laboratory for Advanced System Software, Sogang University Computer Science and Engineering Department. His research interests include cloud computing, cluster-scale deduplication, parallel, and distributed file systems.



Sudharshan S. Vazhkudai is the director of the Hyper-Scale Data Center Program, Computing and Computational Sciences Directorate, Oak Ridge National Lab (a U.S. DOE Lab). In this role, he leads an initiative to build and deploy scalable, distributed storage infrastructure and rich data/metadata management services to capture and support mountains of scientific data emanating from computer simulations, experiments and observations conducted at the various ORNL facilities. In addition, he also contributes to the co-design and deployment of supercomputers and associated solutions for the Nation's Premier Supercomputing Center, the Oak Ridge Leadership Computing Facility (OLCF). OLCF is home to the world's No. 1 supercomputer, Summit, the future Frontier exascale system and the fastest storage system, Spider, providing billions of core hours to a scientific user base from academia, government and industry, to perform breakthrough research in science. Prior to his current role, he led the Technology Integration (TechInt) Group, building solutions for supercomputers in several areas such as file and storage systems, non-volatile memory, data management, system architecture, networking, and distributed systems. He is also a distinguished scientist with ORNL and studies the fundamental underpinnings of supercomputers and data centers in many of the aforementioned areas.

Seung-Hwan Lim received the bachelor's and master's degrees, both from Seoul National University, Seoul, South Korea, in 1998 and 2000, respectively, and the PhD degree in computer science and engineering from Penn State, State College, Pennsylvania, in 2012, under the guidance of Dr. Chita R. Das. Between PhD and master's degree, he worked as a software engineer with Samsung Electronics from February 2000 to August 2005. He joined Oak Ridge National Laboratory in February 2012 as a postdoctoral research associate, transited into a staff member in September 2013. His current research focuses on data analysis methods and systems.



Ali R. Butt received the PhD degree in electrical and computer engineering from Purdue University, West Lafayette, Indiana, in 2006. He is a professor of computer science with Virginia Tech. He is a distinguished member of the ACM. He is a recipient of an NSF CAREER Award (2008), IBM Faculty Awards (2008, 2015), a VT College of Engineering (COE) Dean's Award for "Outstanding New assistant Professor" (2009), an IBM Shared University Research Award (2009), and NetApp Faculty Fellowships (2011, 2015). He was named a VT COE faculty fellow in 2013. He was an Academic visitor with IBM Almaden Research Center (Summer 2012) and a visiting research fellow with the Queen's University of Belfast (Summer 2013). He has served as the associate editor of the *ACM Transactions on Storage* (2016-present), the *IEEE Transactions on Parallel and Distributed Systems* (2013-present), the *Cluster Computing: The Journal of Networks, Software Tools and Applications* (2013-present), and the *Sustainable Computing: Informatics and Systems* (2010-2015). He is an alumni of the National Academy of Engineering's US Frontiers of Engineering (FOE) Symposium (2009), US-Japan FOE (2012), and National Academy of Science's AA Symposium on Sensor Science (2015). He was also an organizer for the US FOE in 2010. His research interests include distributed computing systems, cloud computing, file and storage systems, Internet of Things, I/O systems, and operating systems. At Virginia Tech, he leads the Distributed Systems & Storage Laboratory (DSSL).

Youngjae Kim received the BS degree in computer science from Sogang University, Seoul, South Korea, in 2001, the MS degree in computer science from KAIST, Daejeon, South Korea, in 2003, and the PhD degree in computer science and engineering from Pennsylvania State University, University Park, Pennsylvania, in 2009. He is currently an associate professor with the Department of Computer Science and Engineering, Sogang University, Seoul, South Korea. Before joining Sogang University, he was a R&D staff member with the US Department of Energy's Oak Ridge National Laboratory (2009–2015) and an assistant professor with Ajou University, Suwon, South Korea (2015–2016). His research interests include distributed file and storage, parallel I/O, operating systems, emerging storage technologies, and performance evaluation.