Spring 1-31-2014

Providing Flexible File-Level Data Filtering for Big Data Analytics

Lei Xu
University of Nebraska-Lincoln, lxu@cse.unl.edu

Ziling Huang
University of Nebraska-Lincoln, zhuang@cse.unl.edu

Hong Jiang
University of Nebraska-Lincoln, jiang@cse.unl.edu

Lei Tian
University of Nebraska-Lincoln, tian@cse.unl.edu

David Swanson
University of Nebraska-Lincoln, dswanson@cse.unl.edu

Providing Flexible File-Level Data Filtering for Big Data Analytics

Lei Xu, Ziling Huang, Hong Jiang, Lei tian
Department of Computer Science and Engineering
University of Nebraska-Lincoln
{lxu,zhuang,jiang,tian}@cse.unl.edu

David Swanson
Holland Computing Center
University of Nebraska-Lincoln
dswanson@cse.unl.edu

***Abstract*—The enormous amount of big data datasets impose the needs for effective data filtering technique to accelerate the analytics process. We propose a Versatile Searchable File System, *VSFS*, which provides a transparent, flexible and near real-time file-level data filtering service by searching files directly through the file system. Therefore, big data analytics applications can transparently utilize this filtering service without application modifications. A versatile index scheme is designed to adapt to the exploratory and ad-hoc nature of the big data analytics activities. Moreover, VSFS uses a RAM-based distributed architecture to perform file indexing. The evaluations driven by three real-world analytics applications demonstrate VSFS' high scalability and data-filtering capability.**

I. Introduction

The large volumes of big data datasets, such as system logs [62], web clicks [8], online transactions, DNA databases, scientific experimental data [3], demand data analytics applications to be capable of effectively and efficiently *filtering* data before analyzing them [6], [69]. Additionally, the high variability characteristic of such datasets and the explorative nature of many data analytics activities suggest the strong needs for a scalable and flexible data filtering capability in data-management infrastructure [43].

Traditional RDBMS databases and file systems are two ends of the spectrum of big data management infrastructures, where NewSQL databases [4], [21] and NoSQL databases [13], [16], [24] fall somewhere in between. While storage systems that fall close to the RDBMS end provide rich sets of data manipulating functionalities, which include data filtering, file systems [11], [33], [64] usually offer significantly better scalability and performance, as well as very little restriction on the forms of data [36]. Meanwhile, the big data datasets are usually unstructured or not well-defined when the data were generated, making it difficult to design the schema for database solutions in advance [3], [43], [44]. Therefore, a large fraction of existing analytics frameworks (e.g., MapReduce and scientific computing) are still using file systems to access raw data, for the sake of scalability, performance, and flexibility.

Compared to the NoSQL or NewSQL solutions that attempt to improve the scalability of databases, enhancing the data filtering capability for file systems is more applicable for many big data analytics applications. Traditional file-system based filtering solutions, including various file search engines [27], [34], [42], [49] or those using general-purpose databases as file metadata servers [25], provide the data filtering functionality by searching files to some extends. However, they suffer from poor scalability, low performance, inflexibility and inadaptability, as we will elaborate in Section 2.2.

In this paper, we propose a new form of file system, *Versatile Searchable File System* (**VSFS**), that is designed to offer advanced file-level data filtering functionality to accelerate a class of big data analytics applications. This class of applications share a common pattern typified by the following scenario: *By applying a set of coarse-grained (file-level) rules to filter out (in) the unrelated (related) input files to an analytics application, the scale of the required input data to process can be drastically reduced, resulting in significantly accelerated analytics applications*.

To efficiently support such big data analytics applications [19], [46], [51], [59]–[61], VSFS offers the following salient features that differentiate it from the aforementioned solutions:

Namespace based file query language. VSFS introduces a novel Namespace-based File Query Language (NFQL) that is compatible with POSIX hierarchical namespace [2]. NFQL enables the existing big data analytics applications to directly filter file data *without any code modification*.

Versatile and near real-time file indexing. A flexible index mechanism is introduced to create and customize file indices to support NFQL. Its near real-time file indexing enables high-accuracy file filtering, because all data is updated and none is left un-indexed.

In this paper, we make the following technical contributions:

1) A new file system model is proposed for big data analytics, i.e., a versatile searchable file system, and the proof of its necessity and feasibility. A comprehensive discussion is provided on the design principles of such a file system and its possible interfaces.

2) The prototype development of VSFS in a distributed environment with distributed RAM-based file index storage, its novel file system namespace and index scheme.

3) Extensive evaluations driven by real-world datasets demonstrating that VSFS significantly outperforms a number of large-scale state-of-the-art database-based solutions (i.e., MySQL cluster, HBase, MongoDB and VoltDB) on file-indexing ($12 \sim 4709\times$) and file-search ($5.2 \sim 6275\times$) performance at a reasonable and acceptable I/O overhead.

Finally, we choose a wide range of analytics applications (e.g., Molegro Virtual Docker [60], Distalyzer [51] and Hive [61])) to demonstrate that VSFS is able to speed up their computation dramatically ($30 \sim 1779$ times) by applying coarse-grained filtering.

The rest of this paper is organized as follows. Section II presents the necessary background to motivate the research on VSFS. The design and implementation of VSFS are described in Section III. We evaluate the scalability, performance and effectiveness of the VSFS prototype in Section IV. Section V concludes the paper with remarks on directions of future research on VSFS.

II. Background and Motivation

This section presents the necessary background and elaborates on our observations that help motivate the VSFS research.

A. Need for File-Level Data Filtering

File systems have been considered as the high-performance and large-scale data management infrastructure for big data analytics applications [1], [3], [10], [43], [60], because they provide performance features that are by and large unmatched by many database solutions [17], [36], [54], [58]. As a result, a large number of such analytics applications and frameworks [3], [10] have been built to run on top of various file systems [11], [33], [64], [66].

Because of the huge volumes of big data datasets, the ability to filter out unwanted data has become increasingly critical for the big data analytics applications to complete the computation in near real-time [6], [52], [69]. For example, interactive analytics applications often exploratorily analyze over a small fraction of the big data dataset to obtain immediate feedback and insights [48], [69]. For some applications, it is even impossible to analyze the datasets in the required time without filtering out most of the raw data [3], [70]. Therefore, within the scope of file-system-based analytics applications, We argue that, compared to fine-grained record-level filtering [31], [52], [69], coarse-grained file-level filtering is a less accurate but much faster and potentially sufficient filtering technique, because it is widely applicable to the existing analytics applications' workflows and data models that are file-system based and keeps the system overhead low. We will provide evidence in support of this argument throughout the paper, culminating with a detailed demonstration and evaluation in Section 4.

We present three file filtering methods with their corresponding suitable applications and datasets to give the readers an idea of how the proposed file-level filtering may apply.

Accurate file filtering returns the exact set of qualified files based on the given filtering conditions. It requires each file to contain exactly one complete data record. However, such files can be very large in size, so that it is not suitable for them to be stored within a database [3], [36], [59], [70]. For instance, Molegro Virtual Docker (MVD) [60], a computational drug-discovery application, stores the full structure information of a particular protein in one input file. The MVD application frequently computes a small set of proteins that share similar characteristics to evaluate the effectiveness of a new drug. Not only is the protein-structure dataset typically very large ($(10M \sim 100M)$ files), but also there are hundreds of different attributes from each protein (i.e., structures or energy characteristics). Accurate file-level filtering can quickly filter in the desired files while filtering out unwanted files, thus significantly improving the drug discovering process.

Approximate file filtering with possible false negatives returns a small subset of files that might lead to erroneous results with a small probability due to false negatives (i.e., useful data may have been filtered out). For many big data datasets, such as web index, social network data and machine learning data, the results of analytics are usually probabilistic in nature and obtaining absolutely accurate results is either unnecessary or cost-ineffective [6], [18], [35]. For this kind of applications, such as the machine-learning based log analytics program Distalyzer [51], it would be useful and cost-effective to index files with the extracted features, then generate a sampling plan based on these indexed features to filter in a small set of files, where analysis based on the much smaller, filtered dataset can very expeditiously generate approximate results with a certain confidence.

Approximate file filtering without false negatives returns a small subset of files that *might* contain unuseful data, although no useful data have been filtered out. This filtering method fits well for a wide-range of *ad-hoc* queries on the big data datasets [43], [48], [61], [62], as well as many other algorithms that require partial scanning of the datasets, such as aggregation algorithms [32]. *Hive*, as a concrete example, is a data warehouse system that translates an SQL-like query into a MapReduce job that scans the files stored in HDFS [61], [62]. The *ad-hoc* queries in *Hive* usually need to scan a fraction of the dataset to obtain aggregated values. This behavior makes it possible for *Hive* to benefit from the approximate file-level filtering without false negatives.

The need for file-level data filtering makes it highly desirable for a file system designed for big data analytics to be capable of offering 1) high performance and scalability in data-intensive environments, 2) file-indexing schemes for file filtering that are flexible and adaptive to a variety of data types and big data analytics activities, and 3) ease of use and transparency for a wide range of existing big data analytics applications. In the next subsection, we will examine the existing data management solutions and discuss the necessity and motivation of the VSFS research.

B. Background on Data Management Solutions

The long established file system hierarchical namespace [2] was designed for efficient organization of files but not necessarily for flexible retrieval/filtering of files [26], [29], [47]. More specifically, the static and sole representation of the file path severely limits the room to embed sufficient amount of keywords for retrieval purposes and is unable to dynamically adapt to various complex file-retrieval requests.

To this end, dozens of solutions have been proposed to address the inadequacy of file systems in fast file retrieval and filtering, to some extent. Although these solutions are reasonable techniques within their own targeted application scenarios, they have critical limitations in the big data environments. We broadly divide them into the following three categories:

File search engines [12], [28], [49], which rely on the crawling process to catch up with new updates periodically, are less likely of keeping the file index always up-to-date [42], [68]. This might lead to inaccurate retrieval results. As a result, many data analytics applications could not rely on file search engines to filter out files. Furthermore, none of the existing file-search engines is designed for large-scale data-intensive systems [3], [44], [70].

Database-based metadata services use databases (SQL, NewSQL or NoSQL) as a supplementary file metadata management service running on top of file systems [59]. On the one hand, the SQL or NewSQL solutions must maintain the ACID property of SQL executions, which introduces significant performance overhead [42]. Additionally, the static and stable SQL schema is not well suitable for the explorative and ad-hoc nature of many big data analytics activities [43], [55]. On the other hand, the NoSQL-based solutions, while designed with the scalability goal in mind, either have over-simplified, inflexible data models that are not suitable for multi-dimensional file indices, or suffer from relatively low I/O performance [13], [16], [53]. It is a well-accepted fact that databases are not a "one-size-fits-all" solution [42], [58].

In addition, solutions in these two categories also share some common drawbacks. First, most filesystem-based big data analytics applications contain substantial amount of legacy code, making it difficult and expensive for them to adapt to the new services. Second, as there are no standard APIs for file search or filtering services, it is difficult to migrate the applications from one environment to another. Third, the separation between the file system and the file index will easily lead to an inconsistent state.

Moreover, a common I/O pattern for many big data applications is that the applications treat the entire directory as the input dataset (e.g., Hadoop). To support these applications, file-search solutions need to create a new directory on-demand and place the resultant files into it. However, copying the files into the new directory will lead to significant amounts of data movement and replicated data, and moving (i.e., renaming) the files into the new directory breaks the integrity of the original dataset [43]. While creating hard/soft links is an alternative approach, these links have their own limitations. For example, creating links on a per-file-search basis can significantly complicate the file system namespace.

It is clear that the external file-search services have their own weakness for wide use in the big data environments. We argue that it is a must to offer flexible, advanced file search and filtering functionalities directly from the file systems.

Searchable file system interfaces provide file search functions directly through the file systems. Research projects that attempt to provide such interfaces include Semantic File System [26], HAC [29] and WinFS [50]. Unfortunately, all of these systems are designed to serve the end-user's needs for retrieving files. As a result, they will try to find the files based on the contents in the form of *keywords* provided by the end users, along with very limited support for the metadata query [34], [42]. These queries may not be meaningful for many big data analytics applications that heavily rely on range queries or multidimensional queries to fetch the desired data. Furthermore, similar to the file-search engines, these systems provide a set of pre-defined file content parsers and perform the parsing within the systems, which clearly limits the flexibility in handling the very high variety and heterogeneity of big data. This is because 1) each data analytics application has its own set of file attributes to be indexed, and 2) the attribute values may be generated during the analytics process and thus cannot be written into the original input file (i.e., breaking the integrity of the original dataset) [55], [69]. For instance, the MVD application calculates the energy between molecules and proteins, where the input files are immutable and the energy values must be indexed into the corresponding protein files. As a result, scanning either the original input file or an output file written with energy values can not create the correct record that connect the energy value to the original input file. In other words, these proposed searchable file system interfaces are not powerful, flexible and general enough to provide the required file-filtering functionalities for many of the big data analytics applications.

III. Design And Implementation

To offer adequate file-level data filtering, we present a distributed *Versatile Searchable File System*, VSFS, from the ground up, which provides a high-performance and flexible file search capability for big data analytics applications. VSFS' two core concepts, namely, the POSIX Namespace-based File Query Language (NFQL) and the versatile real-time file indexing, are presented in Section 3.1 with a focus on the versatility and adaptability. Section 3.2 describes the RAM-based distributed architecture that enables the file system level real-time file indexing and search capability. This is followed by a discussion of an example and suggested usage of VSFS.

A. Namespace-based File Query Language and Versatile File-Indexing Scheme

The most unique feature that differentiates VSFS from other file systems is its flexible Namespace-based File Query Language (NFQL), which is deliberately designed to be backward-compatible with the existing POSIX file systems [22], [33], [64], [66] so that *the legacy applications are able to run and search on VSFS without any modification*.

NFQL. VSFS inherits the concept of *using the POSIX directory semantics to perform file-search* [26], [50]. In addition, because VSFS' NFQL is deliberately designed to support analytics applications, its semantics is more flexible and richer than the content-based queries in the existing schemes [26], [29], [50]. As an example, a user of the MVD application

can perform a rather complex operation of data filtering to filter in "*all protein structure files that satisfy the conditions of 1) located under the directory "/foo/bar" (including its sub-directories); 2) energy targeted at "drug-A" is greater than* 10.5 *eV; and 3) weights smaller than* 16 *kilodaltons*" by simply scanning the following directory in NFQL:

"/foo/bar/?drug-A:energy> 10.5&weight< 16/"

Such a file query directory is composed of one *prefix* directory (e.g., "/foo/bar" in the previous example) and one or more *query expressions*. Similar to the programming language and SQL language designs, the query expressions in NFQL are constructed into a single Abstract Syntax Tree (AST). VSFS then creates the query plan based on this AST. Currently, VSFS only supports a limited set of expressions and any combinations of them by using AND ('&'), OR ('|') or parentheses. A simplified NFQL specification is listed in Grammar 1. In an NFQL query, the *prefix* directory must be a physically existing directory in VSFS. The multi-dimensional query uses a form that is similar to a zero-based array in programming languages to access a particular dimension of one indexed attribute (e.g., "$coord[2] > 10$" represents the condition that the third dimension of "coord" is larger than 10). Finally, the top-k query is carried out by using a suffix of "#*num*" with optional '+' or '-' to specify the results in the ascending order or descending order, where *num* indicates the number of records to return.

⟨*query*⟩ := ⟨*prefix*⟩ '/?' ⟨*expression*⟩ [⟨*topk*⟩]
⟨*expression*⟩ := ['('] ⟨*expression*⟩ [')']
| ⟨*expression*⟩ {('&' | '|') ⟨*expression*⟩}
| ⟨*range_query*⟩ | ⟨*point_query*⟩ | ⟨*multi_dimensional_query*⟩
⟨*range_query*⟩ := ⟨*index*⟩ ('>' | '>=' | '<' | '<=') ⟨*value*⟩
⟨*point_query*⟩ := ⟨*index*⟩ '=' ⟨*value*⟩
⟨*multi_dimensional_query*⟩ := ⟨*index*⟩'['⟨*num*⟩']' ('>' | '>=' | '<' | '<=') ⟨*value*⟩
⟨*topk*⟩ := '#' ⟨*num*⟩ ['+'|'−']

Grammar 1. A simplified NFQL specification in the Extended Backus-Naur Form (EBNF).

VSFS treats the query as a dynamically generated file system directory and fills it on-demand with the symbolic links to the actual files that satisfy the query. Because it is dynamically generated, this virtual directory is only visible to the client who issues the query, where the original dataset is not changed [45], [69] and no data movement is incurred [43].

Attribute	Typical Value	Description
root	"/foo/bar"	the starting point (directory) of this index.
name	"energy"	an arbitrary string to identify this index.
index type	range	the data structure used for the index.
key type	floating-point	the data type used as the key.

TABLE I
AN EXAMPLE OF FILE-INDEX DEFINITION FOR PROVIDING RANGE QUERY ON FLOAT "ENERGY" VALUES.

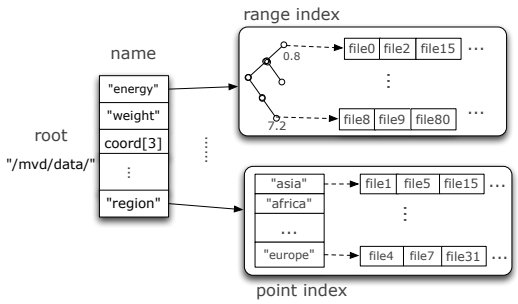

Fig. 1. An Example of the Logical View of VSFS' Versatile File-Indexing Scheme.

Versatile File-Indexing. In order to support NFQL, VSFS provides a *versatile file-indexing* scheme that allows users and analytics applications to create *arbitrary file indices anywhere within the file system* on the fly. Therefore, users and analytics applications do not need to be confined to the pre-determined index schemas (e.g., SQL tables). A file index is defined by a four-parameter tuple ***(root, name, index type, key type)***, where the first two parameters **(root,name)** provide its unique identification. In the current implementation, each index maintains a one- or multi-dimensional key-value structure, in which all keys in the same index share the same type (e.g., int or string), and the values are a set of file identifiers. A logical view of VSFS' versatile file indices is illustrated in Figure 1. We describe these four index definition parameters in more details next.

First, *root* is the path of the top directory of the namespace covered by this file-index, which means that only the files under this directory and its sub-directories can be indexed into this index. By specifying the "*root*" of a file index, VSFS effectively provides the controllability over *the scope of a file index*. As a result, users can specify a file index for one particular dataset under one directory, instead of the entire file system [12], [26], [42], [50]. Moreover, this notion of "file index scope" not only enables many practical usages in the big data applications (e.g., the MVD and Hive examples in Section 4.4), but also explicitly defines the boundary between the file indices. Knowing the explicit boundary of file indices is critical to implementing a feasible real-time file indexing architecture in a large-scale system, because it effectively confines the indices to a relatively small scope[34], [42]. It is worth mentioning that an NFQL query will conduct queries on all indices that are under the "prefix" directory.

Second, *name* is a descriptive string of the index. It allows the users to specify an arbitrary number of customized indices, each of which has a different name, on the same dataset (i.e., using the same "root"). Therefore, users can associate any attributes of interests to their files, which are not limited to file metadata [34], [42] or to the pre-supported attributes [12], [27]. Additionally, the indices with different names are independent from one another, which means that a file does not need to have records in all indices.

Third, *index type* describes the desired performance and functional characteristics of the file index, which is used by

VSFS to choose the appropriate data structure for this file index. Currently, VSFS supports three index types: "*range*" (B-tree) for range query, "*point*" (Hash Table) for point query and "*multidim*" (K-D-Tree [15]) for multi-dimensional range query.

Forth, *key type* describes the data type used for the key of an index. VSFS currently supports the data types of integer, floating-point, and string. It is important to provide such choices for the users because in the big data environment there are various demands to store different key types.

Finally, each file index has the UNIX file permission, i.e, uid (*user id*), gid (*group id*) and mode (*UNIX permission*). Therefore, users can control the access to file index in the same way as controlling accesses to regular files or directories.

With the highly customizable file indices, the applications *should* and *must* have the responsibility of feeding the contents of indices of their files, instead of letting the file systems parse and index the files [12], [26], [50]. While this may appear to be a nontrivial burden for end-users, *it actually offers the analytics applications the necessary flexibility to decide when and what to index, or even allowing these applications to perform in-situ file indexing* [43], [44], [63], [70]. It is these design choices that significantly differentiate VSFS from the existing systems [12], [13], [26], [34], [42], [50].

B. RAM-based Distributed Architecture

VSFS is a distributed file system designed to include two near real-time capabilities for data-intensive environments, namely, file-indexing and file-search. To this end, we design a unique RAM-based distributed architecture that enables VSFS to offer such capabilities. For the management of file data, our current design chooses to delegate this responsibility to the existing matured storage systems and abstract them to an ObjectStorage interface for VSFS to use [7], [33], [56], [64], [65]. The traditional distributed file system issues, such as I/O performance, durability, data placements, etc., are managed by these underlying storage systems. With this design choice, the VSFS development is significantly simplified while the advantages of traditional file systems are preserved in VSFS.

Next, we discuss the design of VSFS' metadata and index management infrastructure that supports the real-time file-indexing and -search capabilities. VSFS is built on a RAM-based distributed metadata and index architecture, which is designed to scale to hundreds of metadata and/or index servers to support tens of Terabytes of file indices for tens of Petabytes of raw data.

In its entirety, a metadata and index cluster of VSFS consists of two Consistent Hashing (C.H) Rings [24], [38], [39], respectively, where one C.H ring is constructed by *Master Servers* and the other is constructed by *Index Servers*. Additionally, analytics applications can use either VSFS' API or FUSE-based VSFS client to access the VSFS' file-filtering service and the object store, as shown in Figure 2.

Master Server takes charge of the namespace for both files and indices, as well as the file metadata. Namespace and metadata are distributed to the master server cluster's

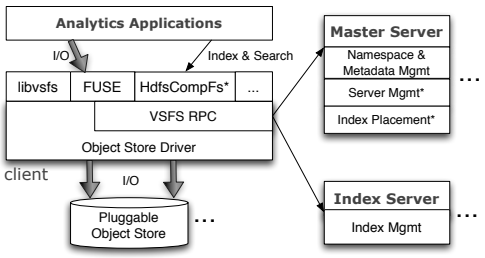

Fig. 2. The VSFS Architecture. (*)HdfsCompFs is an HDFS compatible file system under development. (*)Server management and index placement are only managed by the primary master server.

C.H ring. A "primary" master server is chosen from the master server cluster by using a leader election algorithm. Its additional responsibility is to manage all master servers and index servers. The single primary master server simplifies the clients' route requests. We only keep one copy of the cluster topology information in the system, and thus it is easier to guarantee the consistency. Moreover, the cluster topology information is aggressively cached on the client side, which prevents the primary master server from being an access bottleneck, as implied by the evaluation results of Sections 4.1 and 4.2. Compared to the regular IOs and file-indexing/search IOs, the changes in the cluster topology, due to node addition/deletion or load balancing, are rare in the system. Therefore, it is reasonable to propagate the cluster topology changes to the object store synchronously. When the primary master server fails, a newly elected master server quickly reestablishes the cluster topology by loading the saved cluster typology information from the persistent object store.

Object Identifier. Each file in VSFS is assigned a 64-bit system-wide unique *Object Identifier* (oid), which is used to identify a file in an index server. However, it imposes a unique requirement for VSFS metadata organization: VSFS must efficiently map between file path and oid in both ways, because clients need to fast resolve the file paths from the oids queried from the index servers. This is why VSFS uses its own master servers to take over the responsibility of file system namespace management. Furthermore, to improve the scalability and the capability of resolving file paths in parallel, master servers are organized in a C.H ring, where the key of the C.H ring is the hashed file path. Obviously, to fast resolve a file path from the corresponding oid, it requires both the oid and file path of a given file to reside in the same master server. Therefore, we design a pseudo hash algorithm to calculate oid as follows:

$$oid = prefix(hash(path)) + unique\_value$$

where the first 16 bits are calculated from the 16-bit prefix of the hashed file path (e.g., using MD5), and the remaining 48 bits are assigned by a server-wide unique value. It ensures that the oid has the *exact* same distribution as the hashed value of the file path, and it also guarantees the system-wide uniqueness of oid.

VSFS uses this pseudo hashed "oid" to distribute file metadata to multiple master servers, and stores the sub-file

list (analogous to the *directory file* in Ext4) on the same server that stores the directory's metadadta. This means that a file is usually on a server different from the one housing its parent directory. Thus, we use an optimized concurrency control technique, which is inspired by the Optimistic Concurrency Control [40] for operations that need the cooperation of two master servers. For example, creating a file requires two RPCs, where the first RPC creates a new metadata record on the server *where* the 'oid' resides, and the second RPC inserts the file name to its parent directory. Before the completion of the second RPC, this file is invisible to other clients. In the meantime, if two clients attempt to create the same file simultaneously, one of them will fail the first step and then will retry to check the file's existence. Furthermore, this oid generation algorithm makes it easy to rename and delete files. Deleting a file in VSFS will erase the file record in the namespace but will not erase the corresponding index records from the index servers immediately. Background garbage collection threads running on index servers will reclaim the resources occupied by the deleted files. Similarly, renaming a file will link the old "oid" to the newly obtained oid. Thus, the index records associated with the old oid do not need to be updated, and VSFS can return consistent file-search results without losing any index record associated with the old file path.

Index Server manages various kinds of file indices and answers client's index/search requests. To achieve the lowest-possible file-indexing and -search latency, each index server keeps all file indices within its RAM, and applies common file system techniques, such as write-ahead-log [30], to offer the durability of file index by recording them into the object store. Periodically, the index servers write one in-RAM index as an index image to the object store, and discard all the corresponding logs to save space. Since flushing logs is not in the I/O critical path, it is not included in the file indexing latency.

In order to flexibly support the aforementioned versatile file indices, an index server manages the index metadata, including *index type* and *key type*, besides the index structure itself. Since the index server is the only one in the system that has the knowledge of the key type of a particular index, the client sends the index keys in the form of a string and lets the index server interpret the keys.

Furthermore, to scale and balance large file indices, as well as explore the potential parallelism for index accesses, a large index is divided into smaller *Index Partitions*. If the scale of a partition exceeds a certain threshold, it will be divided and live migrated to one of the other index servers [9], [20]. All the partitions of the same index are organized as one logical C.H ring that maps file oids to a particular index partition, where the starting point of the key range in each partition is chosen as its *partition ID*. This per-index logical C.H ring is also managed by the primary master server. In addition, to easily locate the actual index server that is responsible for a partition, VSFS organizes all index servers into a large C.H ring as mentioned earlier, where the key is calculated by the hashed value according to the combination of $(root, name, partition\_id)$. As a result, VSFS can statistically balance index partitions globally.

In the current stage of the VSFS development, the focus is on exploring the design space for a scalable high-performance file-search facility and its impacts on the existing analytics workflows. Nonetheless, VSFS provides a basic level of failure tolerance capability for a failure of any single node within one C.H ring, whether it is the master server C.H. ring or the index server C.H. ring. The persistent data of each C.H node is stored in the object storage, therefore, when a node on C.H ring fails, its adjacent node loads the data (e.g., file metadata or index) from the shared object storage and serves the requests. The reliability issues of Consistent Hashing-based distributed systems have been extensively studied [9], [14], [24]. VSFS will benefit from the existing reliability work on Consistent Hashing and implement advanced reliability features as its future work.

VSFS Client parses the requests from the analytics applications through VSFS' API or the file system client implementation (e.g., FUSE), creates a query plan, and issues the requests through RPCs to the VSFS cluster. Currently, we use FUSE to implement a POSIX-compatible file system client for VSFS. In addition, both the master server C.H ring and the index server C.H ring are aggressively cached in the FUSE daemon. Therefore, in general, the clients do not need to communicate with the primary master server to route the metadata or index operations to the servers. This C.H ring cache is updated only if there are metadata or index requests that have failed to be delivered.

File Indexing Workflow. The VSFS client first uses the *(name, file path)* pairs to ask the primary master server to locate the index partitions to which these files belong. If a client inserts file index records in batches, it will first pre-process all the file paths in the same batch and obtain the parent directories shared by these files, and then use the *(name, parent path)* pairs to locate the index partitions. This optimization is usually quite efficient in the big data environments, because the analytics applications tend to index files in the same dataset (i.e., placed in the same directory). With the obtained partitions and the cached topology information of the index server C.H ring, the client is able to use the partition IDs to find the corresponding index servers locally. Then the client transforms the index keys into the string form, and sends the records as *("key", oid)* pairs to the corresponding index servers. It is also worth mentioning that all file-indexing operations are triggered independently of the raw file IOs, therefore their latencies are not included on the I/O critical path.

File Search Workflow. We use the FUSE-based VSFS client to illustrate how VSFS accomplishes a file search. When an analytics application searches files by issuing a "readdir()" system call on an NFQL virtual directory, the FUSE daemon passes this directory to the VSFS client. The VSFS client parses this virtual directory and creates an AST based on the NFQL specification. The client communicates with the primary master server to obtain the metadata of all potential

index partitions that *might* have the resultant files. The VSFS client again uses the locally cached C.H ring to route the search requests to the corresponding index servers. These index servers return oids, which the client uses to find the master servers that house these oids. File-path-resolving requests are sent to these master servers in parallel to obtain the resultant file paths. Finally, VSFS fills the virtual directory in FUSE with the symbolic links to the resultant files. If there is no file metadata for a dangling oid returned from an index server, it means that this oid has been deleted and therefore it is omitted from the resultant files.

With our design, the RAM-based distributed architecture allows VSFS to deliver very high file-indexing and file-search performance as well as high scalability.

C. VSFS APIs and Example Usage

Besides the FUSE-based implementation, VSFS also offers a set of native APIs in C++ for analytics applications to directly manipulate indices and issue queries, as illustrated in Code 1.

```
int create(const string& root, const string& name,
    int index_type, int key_type);
int destroy(const string& root, const string& name);
int update(const string& name, const string& file,
    const string& key);
int remove(const string& name, const string& file,
    const string& key);
int search(const string& nfql, list<string>* results
    );
```

Code 1. VSFS' C++ APIs.

Additionally, VSFS provides users with a command-line tool (vsfs) to manipulate indices. Through vsfs, users can specify the parameters to customize file indices. It also supports the UNIX pipeline to feed the index records. Therefore, an analytics application only needs to print the index records to the stdout in the format of "*filepath name key*\n" for each record and vsfs will route the requests to the corresponding indices. To illustrate how effortless it is to integrate VSFS into an existing application (i.e., MVD) running on an HPC resource management system (SLURM [41]), we show the following example:

```
#!/bin/sh
# Create indices on directory ``/data''
vsfs index create /data ``drug-A:energy'' --index
    range --key float
vsfs index create /data ``weight'' --index range --
    key uint32
# Populate the index through a UNIX pipeline.
srun mvd /data | vsfs index insert ``drug-A:energy''
# Run against the search resultant files.
srun mvd ``/data/?drug-A:energy>10.5&weight<16/*''
```

Code 2. SLURM Job Using VSFS.

To efficiently utilize VSFS, the granularity of the data entity should be considered. In theory, dividing data into finer-grained segments can guarantee high file-level filtering accuracy. However, this is considered a less appropriate practice in VSFS, because it will result in a huge number of small files, along with their index records. For example, given a 1024×1024 pixel GIS graph, where each pixel has 200 attributes, it is not advisable to index 200 attributes for each pixel, as it will generate $200M$ index records. Instead, by indexing 64×64 pixel blocks of the original graph, the scale of the index can be reduced by $4,096$ times, while still allowing for effective shrinking of the dataset by up to 256 times by filtering out unwanted areas in this particular example. The best ratio between the sizes of index and raw data is application dependent and should be determined by domain experts, a topic that is beyond the scope of this paper. We encourage users to use VSFS as a supplementary coarse-grained "*pre-processing*" filtering service for the existing applications (e.g., SQL engine, Scientific Computing [3], [60], [70] or MapReduce [31]). Additionally, because of VSFS' capability of dynamically creating indices on the fly, it is not necessary for the application to create and feed all indices at once, or *in-situ* [43], [44], [62]. Users can build and destroy file index on-demand for real-time interactive analytics on big data datasets. Finally, with appropriate access control, users from different groups can potentially create indices that are invisible or inaccessible to others on the same dataset. This is an important feature for a multi-user collaboration environment [43], [55].

IV. Evaluations

We evaluate the performance of the VSFS prototype using representative datasets, workloads and applications. In the experiments, we examine the performance in terms of *file-indexing* performance, *file-search* performance, *I/O performance* and *application performance*.

Experimental Setup. We prototype VSFS on a 20-node heterogeneous Linux cluster testbed. Each node features $1 \sim 2$-socket Quad-core AMD Opteron 2354 2.2GHz or Dual-Core AMD Opteron Processor 2220 2.8GHz CPU with 8GB RAM. These testbed nodes use 1Gb Ethernet to connect to a Dell Force10 S50N 48-port 10GbE switch that in turn links to a production HPC cluster through a 10GbE Fiber. Each node runs Scientific Linux 6.3 and uses the local disk (60GB) to store the experimental data. We compare VSFS' file-indexing and file-search performance against the MySQL Cluster 7.2.10, HBase 0.94.6, MongoDB 2.5.1 and VoltDB 3.4, because they represent the current state-of-the-art file management solutions in large-scale systems. We also evaluate VSFS' raw I/O performance. Finally, we use three data analytics applications to demonstrate their performance gains from using VSFS.

To make the performance comparison as fair as possible, we optimize the MySQL/HBase/MongoDB/VoltDB clusters to the best of our knowledge. All clusters are configured to be auto-sharding or partitioned based on the hashed file path. To achieve the best write performance, we only use 1-replica of data in all testing clusters. All tests use 64-bit integer as the key for indices and the value is the string form of the file path. Moreover, every node is configured to use 80% of its physical RAM buffer to maximize memory utilization, if such a configuration is applicable. Table II summarizes the key features of the schemas and systems used in VSFS' evaluation.

Target System	Category	Index Schema	Characteristics
MySQL(p)	Trad. SQL	One index per SQL table (i.e., partitioned table).	Better Write Performance
MySQL(s)	Trad. SQL	All file indices in the same SQL table (i.e., single table).	Better Query Performance, Inflexible Schema
HBase	NoSQL	One index per HBase table, file path as *rowkey*.	Large-scale, Hadoop-based
MongoDB	NoSQL	Files are documents in the same collection, adding index record as document field.	Flexibility, Popular File Metadata Server
VoltDB	NewSQL	All file indices in the same SQL table since its SQL schema cannot be dynamically modified during runtime.	In-Ram Database, Speed

TABLE II
THE INDEX SCHEMAS AND CHARACTERISTICS OF THE COMPARED SYSTEMS.

A. Index Performance

The objective of this evaluation is to test each targeted system for its ability to handle a large number of file index updates in real time. To stress the targeted systems, we use 30 physical nodes from the production HPC cluster to run 4 client processes per node and each client process sends file-index records to two individual file indices. Therefore, there are a total of 120 clients sending index update requests to a total of 240 file indices. Additionally, we choose 1024 requests as the batch size for all tests. In each test, the clients issue a total of 10 million records.

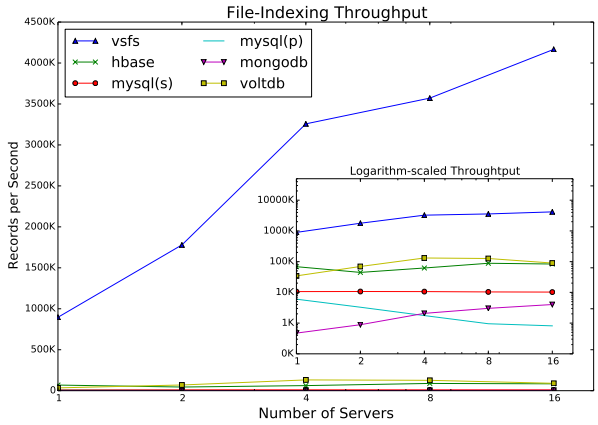

Fig. 3. File-Indexing Throughput.

As illustrated in Figure 3, VSFS scales significantly better than all of the targeted systems. VSFS is $12 \sim 49\times$ faster than HBase, $103 \sim 192\times$ faster than MongoDB, and $24 \sim 47\times$ faster than VoltDB. VSFS outperforms MySQL even more significantly: $85 \sim 408\times$ faster when a single SQL table is used and $1492 \sim 4796\times$ faster when the partitioned SQL tables are used. The reason behind the degraded throughput in the partitioned MySQL cluster is that it needs to perform a prefix matching between the path of a file and the root paths of indices on one meta-table that stores the mapping from the *(root path, index name)* pair to the actual SQL table name. Therefore, the SQL engine node (SQL node) needs to pull data from multiple data nodes to perform this index-table-locating task. When a single table scheme is used, the bottleneck of the MySQL cluster is shifted to the CPU on the SQL node, because all SQL queries must go through this single SQL node as MySQL does not support distributed locking on a table. In HBase, it is similar to the partitioned MySQL cluster case in that it needs to perform the matching for the prefix of file path to find the corresponding table for a particular file-index. MongoDB, as a NoSQL database, scales well in the test, but it suffers from slow update operations. VoltDB, as a RAM-based NewSQL database, directly shows the overhead of ACID when compared to VSFS' index servers, which is also a RAM-based solution. Its throughput even drops after the scale of cluster exceeds 4 nodes because of high overheads introduced by the cross-machine transactions for intensive updates. Finally, because its architecture is optimized for the file-indexing workloads, VSFS has outstanding file-indexing performance compared to the existing solutions.

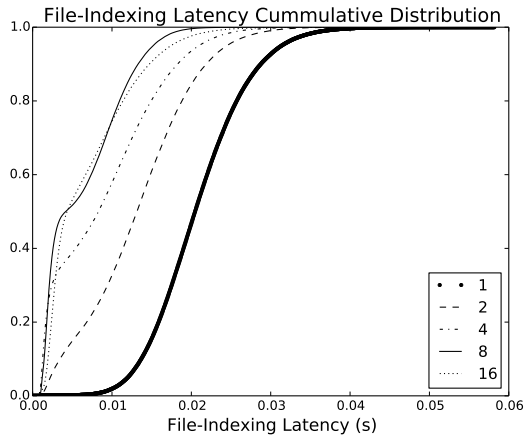

Fig. 4. VSFS File-Indexing Latency Distribution. Each curve represents the file-indexing latency of the VSFS cluster with the given number (1,2,4,8 or 16) of index servers.

The second test takes a closer look at the latency of file-indexing operations in VSFS. In this test, we use 120 clients that simultaneously send file-indexing requests in an open-loop loading fashion. More specifically, every client sends 10,000 index requests, each of which carries a single file-index record. The latency of each request is collected. 4 master servers are used in all the tests, and different numbers of index servers are launched to measure the scalability of the VSFS cluster. As illustrated in Figure 4, the latency of file-indexing operations improves significantly when the VSFS cluster scales up. With more than 8 index servers, 50% of the requests can be completed in 4ms.

In summary, VSFS outperforms the current widely-used SQL, NoSQL and NewSQL based solutions to offer a much better file-indexing performance in data-intensive environments.

B. Search Performance

In this subsection, we evaluate the file-search performance of VSFS, MySQL, HBase, MongoDB and VoltDB. In all tests, 100 file indices, of which each includes 100,000 records, are populated before processing file-search requests.

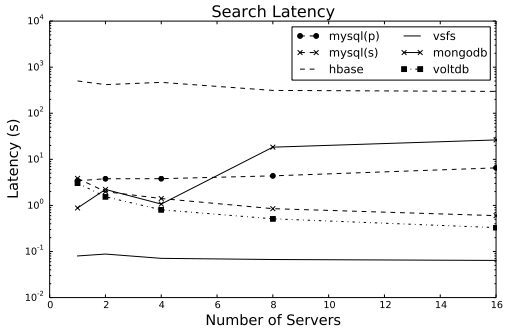

Fig. 5. File-Search Latency as a Function of the Number of Index Servers.

In all tests, a single client issues one file-search request of *"gathering* 10,000 *files from the same index"*, which is a common form of requests in the MVD analytics. The end-to-end latency of the file-query request is measured, as shown in Figure 5. VSFS is up to $49 \sim 6275\times$ faster than HBase and $43 \sim 102\times$ faster than MySQL with partitioned SQL tables and $48 \sim 110\times$ faster than MySQL with a single SQL table, up to $11 \sim 411\times$ faster than MongoDB, and $5.2 \sim 38\times$ faster than VoltDB. This significant performance advantage of VSFS stems from our RAM-based design that enables in-memory processing for VSFS.

Number of Servers	1	2	4	8	16
95*th* Percentile	64	53	52	46	46
99*th* Percentile	507	73	66	69	68
Average	36	19	19	18	19
Peak	674	432	525	385	340

TABLE III
DISTRIBUTION OF STRESS-TESTED FILE-SEARCH LATENCY (MS) ON VSFS.

We conduct the second evaluation to stress test VSFS by issuing simultaneous file-search requests in an open-loop manner. In this test, we acquire 25 physical nodes from the production HPC cluster, where each node runs 4 clients. In each client, 20 threads are created to issue file-search requests. Each thread sends 1000 file-query requests, each of which queries 1000 files from an individual file index in an open loop manner. There is no overlap for targeted indices between any two clients. The distribution of the latency for every request is measured, as listed in Table III. This low-latency file-search operation makes it feasible for VSFS to be deployed directly as a filesystem in data-intensive environments.

C. I/O Overhead

VSFS implements a general POSIX-compatible distributed file system through FUSE on the client machines. VSFS is compared with a production Lustre file system to demonstrate that VSFS indeed incurs acceptably small raw I/O overhead. 8 pre-defined Filebench [67] workloads, including 5 synthetic workloads and 3 metadata-intensive workloads, are used to measure the overall overhead of VSFS. The total number of files in each workload is 100,000 with an average file size of 8MB and an I/O size of 8KB. Each workload is configured to run with 16 threads for 60 seconds. Direct-I/O and big-write options of the FUSE framework are turned on for optimized performance.

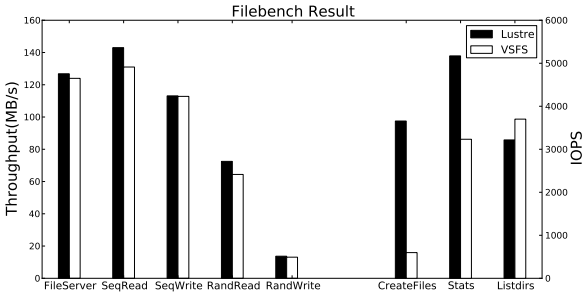

Fig. 6. Filebench Results. VSFS, as a FUSE-based file system, only introduces small additional overhead to the native FUSE overhead. The additional VSFS overhead comes mostly from the file creation and deletion operations.

As shown in Figure 6, for synthetic workloads, VSFS only causes a 3.3%, 8.4%, 0.3%, 11.2% and 4.4% I/O throughput degradation in the *fileserver*, *singlestreamread*, *singlestreamwrite*, *random read* and *random write* workloads, respectively. For the metadata-intensive workload *CreateFiles*, an 83.67% IOPS drop is introduced because file creation will incur two RPCs to two different master servers. However, in big data environments, file creation operations happen far less frequently than other I/O operations [5]. As a result, they will not affect the overall performance significantly, as demonstrated in the *fileserver* benchmark. Even so, we believe that there is still room for optimization and we will leave it as our future work. Also, under workload *Stats* VSFS experiences a 37.5% IOPS reduction. For the *listdirs* workload, which is a very frequently used procedure by big data analytics applications [5], [23], VSFS is actually 15% better than Lustre because this operation is specifically optimized and handled by the VSFS master server.

In summary, the FUSE-based VSFS is shown to incur relatively low I/O overhead to raw data access and acceptable overhead to metadata access, suggesting that VSFS is a feasible distributed file system deployable in big data environments.

D. Application Performance

In this subsection, we apply three different file-filtering strategies to three different applications, namely, Melegro Virtual Docker (MVD) [60], Distalyzer [51], and Hive [61], respectively, to demonstrate the capability and flexibility of VSFS in its support of big data analytics applications. Due to our lack of privileges to mount a FUSE-based file system on the production HPC cluster, all tests in this evaluation run on our 20-node cluster testbed. Due to the limited resource, we configure a 4-node sub-cluster to run the file-search service (1 master server and 3 index servers) to demonstrate a small distributed installation of VSFS, and use the remaining 16

nodes as worker nodes for the applications of MVD and Hive, while Distalyzer, being a local application, runs on one of the 16 worker nodes during the evaluation. It must be noted that VSFS is capable of supporting a much larger number of worker nodes than this small installation, if sufficient compute resource is made accessible.

Molegro Virtual Docker (MVD) [60] represents the category of applications [3], [37] that can utilize the *accurate filtering* strategy. The traditional workaround for these applications is to run a file-search engine or metadata database to search files. Therefore, we compare the use of VSFS as a file-search service against other widely-used file-search solutions, namely, MySQL, HBase, MongoDB and VoltDB.

In this experiment, the MVD program is concurrently running on all of the 16 worker nodes, where each run of the MVD program reads a $\sim$ 4KB file as input, computes the data, writes the output file ($\sim$ 4KB) back to the Lustre file system in the production HPC cluster and generates one record to be indexed. Each original run takes approximately 7 seconds in the production HPC cluster. As a result, to emulate the high workload intensity of simultaneously running the embarrassingly parallel MVD program on a 1000-node HPC cluster that would presumably be $1000/16 = 63$ times faster than the 16-node testbed cluster, we proportionally reduce the computation time of the MVD program by a factor of $1000/16 = 63$.

The evaluation simulates a use case in which the biologists attempt to analyze 10% of the protein data based on the previous runs of the MVD program. In this evaluation, we assume that there are $10,000$ input files in total, based on the numbers provided by domain scientists. Moreover, we assume that there are 500 file indices, of which each contains $10,000$ records that have already been imported to the system. That is, before running the experiments, the $10,000$ file-index records are generated for each index. In the original MVD case (i.e., denoted by "*MVD*" in Table IV), which represents the current practice of the MVD analytics in the real-world environment, the analytics application brute-forcedly runs against all input files. In the other five cases, the MVD analytics application will utilize the external file-search services provided by MySQL (both the "s" and "p" versions), HBase, MongoDB, VoltDB and VSFS respectively to filter out unrelated data. Therefore, in each of these five cases, the file-indexing service first indexes the $10,000$ file records obtained from previous runs and then use range query to filter in $1,000$ targeted files. In the end, the MVD analytics application runs against these $1,000$ files. The execution time of each step is measured.

As illustrated in Table IV, with the help of the external file-search service, the execution time of the MVD analytics can be significantly reduced. However, it clearly shows that most of the comparison solutions add significant indexing and search latencies to the total processing time of the MVD analytics program, while those of VSFS are very insignificant compared to the MVD computation time. In this evaluation, running MVD on VSFS is up to 2.6× faster than the MySQL-based solutions, 1.5× faster than the HBase-based solution, 4.3× faster than the MongoDB-based solution, 1.16× faster than the VoltDB-based solution, 10.3× faster than the original MVD solution. It is worth mentioning that in the MySQL and VoltDB cases with a single table, the latencies of inserting and searching files will dramatically increase on larger data-sets. Moreover, the indexing latencies will significantly increase if multiple attributes are being extracted and indexed for each run, making the MySQL and VoltDB indexing overheads larger than the original brute-forced approach.

Solution	Indexing	Search	Analytics	Total	SLOC
MVD	N/A	N/A	123.600s	123.600s	0
MySQL(p)	18.842s	0.162s	11.800s	30.804s	405
MySQL(s)	8.151s	1.450s	11.800s	21.401s	411
HBase	3.630s	2.615s	11.800s	18.035s	391
MongoDB	1.600s	38.200s	11.800s	51.600s	122
VoltDB	1.740s	0.379s	11.800s	13.919s	167
VSFS	0.127s	0.043s	11.800s	11.970s	0*

TABLE IV
A BREAKDOWN OF THE MVD PROCESSING TIME INTO INDEXING, SEARCH AND COMPUTATION (ANALYTICS) AND THE EXTRA SOURCE LINE OF CODE (SLOC) REQUIRED. (*) MVD DOES NOT NEED TO MODIFY APPLICATION CODE TO UTILIZE VSFS, INSTEAD, THE USERS ONLY NEED TO CHANGE THE PARAMETERS THROUGH THE COMMAND LINE TO RUN MVD.

Distalyzer is a log analytics tool that uses machine-learning algorithms to structurally diagnose performance problems in distributed systems [51]. We use Distalyzer to explore the possibilities of accelerating analytics process by approximately filtering data with possible false negatives, because Distalyzer's source code and datasets are publicly available and it represents typical machine-learning workloads. In this test, we run Distalyzer against its HBase trace (1.3GB) [16], because the authors claimed that the pre-processed HBase trace (902076 requests/55MB) "*avoids performing much analysis on the* 95% *(Hbase) good requests*", which clearly indicates that the HBase trace is filterable. The same pre-processed Hbase trace is split into 184 smaller files, each of which contains 5,000 requests. The counts of high latency events (e.g., $latency > 150ms$) are extracted from each file and indexed into VSFS.

As this test is designed to show the relationship between the application/dataset characteristics and the efficiency of different approximate filtering techniques, Distalyzer runs with three datasets: 1) the unmodified Hbase trace, 2) the VSFS-filtered Hbase trace without false negatives, which is the files that include *all high-latency events*, and 3) the VSFS-filtered Hbase trace with possible false negatives, which is the *single* file that contains *the largest number of high-latency events*. It's worth mentioning that all three traces are based on the same pre-processed HBase trace used in the original work. Moreover, the outputs of Distalyzer analysis are the graphs of the events that represent the causal-chain of root causes of performance problems, as well as the "feature scores" accumulated from these graphs. Therefore, we calculate the output graph similarity by

$$1 - (|E_{diff}| + |V_{diff}|)/(|E_{unmodified}| + |V_{unmodified}|)$$

, where $G(V,E)$ represents the output graph. Finally, the results

of running Distalyzer with the other file-search solutions (i.e., MySQL, HBase, MongoDB and VoltDB) are not illustrated here, because they are consistent with the pattern and trend in the MVD test.

Trace	Execution Time (s)	Graph Similarity	Scores	I/O Reduction
Unmodified	756	100%	3.72	0%
No false negative	785	100%	3.72	97.8%
With false negative	158	87%	3.13	99.4%

TABLE V
THE PERFORMANCE AND ACCURACY OF RUNNING DISTALYZER WITH VARIOUS FILTERING TECHNIQUES.

As illustrated in Table V, it is interesting to see that, by applying the approximate filtering without false negatives, IOs are significantly reduced, yet the analytics process remains slow. This is because Distalyzer builds feature trees based on the high-latency events. Filtering without false negatives means that all high-latency events are preserved, which explains why it has the same output graph and feature scores as the unmodified trace. On the other hand, by allowing false negatives when filtering data, VSFS returns the single file that contains the highest count of high-latency events. As a result, Distalyzer is sped up by 4.78×, while the output graph is 87% similar to the unmodified trace. Additionally, a manual comparison has been applied to the output graphs and it verifies that all vertices and edges that lead to the same problem diagnosis in the original work are preserved in all graphs. The relative order of the vertex weights is also consistent in all graphs. Therefore, we consider the results from the third trace valid.

This test demonstrates that filtering data with false negatives is a viable technique for big data analytics, even for datasets that have already been filtered.

Hive [61] is a data warehouse system that provides easy data summarization and *ad-hoc* queries on large datasets stored in Hadoop-compatible file systems [11]. Typically, Hive translates the user queries, which use a SQL-like query language called HiveQL, into Hadoop MapReduce tasks running on the target datasets. We use Hive to demonstrate a category of applications [43], [44], [62], [69] that can utilize VSFS to *filter data without false negatives* to accelerate computing.

In this evaluation, we use the TrionSort dataset from the Distalyzer test [51]. To achieve the scale of a typical MapReduce dataset [57], the original TrionSort dataset is intensified in this evaluation to achieve a 200GB-scale dataset as follows. Each record in the original dataset is simply replicated 300 times with the timestamps unchanged. Therefore, this dataset intensification does not change the temporal distribution of records [34]. Furthermore, in order to evaluate the impact of file size, i.e., the granularity of indexing, on the computation performance, the dataset is split to contain 1-second, 10-second, and 30-second log records per log file (donated as "1s/file", "10s/file" and "30s/file" in Table VI), respectively. Finally, the intensified dataset is placed in an HDFS directory without being re-organized to favor any form of queries, and two external Hive tables are created in the same directory, where one table is built with the Hive index while the other is not. The reason for using external tables is because it reduces the duplicated data in the system, avoids data movement caused by re-organizing raw-data in favor of given query before running *ad-hoc* analytics operations and provides more flexibility to analyzing the dataset without being bound to a fixed SQL schema [43], [44], [55]. Additionally, all columns that will be used in the HiveQL query conditions are built with Hive indices.

```
/* Create both external tables */
CREATE EXTERNAL TABLE trionsort (time double, type
    string, event string,
attr_name string, attr_value double)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION 'hdfs://node/trionsort';

/* Create index on one table */
CREATE INDEX ts_idx ON TABLE trionsort(event,
    attr_name, attr_value)
AS 'org.apache.hadoop.hive.ql.index.compact.
    CompactIndexHandler'
WITH DEFERRED REBUILD;
ALTER INDEX ts_idx ON trionsort REBUILD;
```

Code 3. HiveQL for building tables and index

In VSFS' current configuration, we run VSFS' master servers and index servers to manage the namespace for HDFS and to answer the file queries respectively. The data-filtering process is done by first running VSFS' command line tool, then asking HDFS to move the targeted files into a new HDFS directory, and finally creating an external Hive table on this HDFS directory. The dataset is first scanned to extract the maximum values of all attributes from each file and index them into the VSFS-managed indices. That is, VSFS creates an individual index for each attribute (i.e., ~ 50 attributes in total), where the index name is the attribute name (attr_name) and the root of the index is the root directory of the dataset in HDFS ("hdfs://node/trionsort").

	Hive+Index	VSFS(1*s*/file)	VSFS(10*s*/file)	VSFS(30*s*/file)
Index granularity	record	file	file	file
# of Records	3635713200	754813	82464	32130
Index size	30GB	43MB	4MB	1.8M
Time to build index	24198s	1320s(t_{par}) + 14.22s(t_{ind})	542s(t_{par}) + 5.32s(t_{ind})	549.1s(t_{par}) + 5.08s(t_{ind})

TABLE VI
CREATING INDICES ON THE DATASET. t_{par} AND t_{ind} DENOTE THE LOG PARSING TIME AND THE INDEXING TIME IN VSFS, RESPECTIVELY.

We first summarize the key characteristics of the task that prepares the index in Hive and VSFS in Table VI. It clearly shows that the coarse-grained indexing results in significantly higher indexing performance at a smaller space overhead. For simplicity, we use a non-distributed Python script to scan and index the VSFS datasets, and it still yields a $3.2 \sim 7.8\times$ speedup over the Java-based distributed MapReduce task (Hive) on a 16-node Hadoop cluster.

To evaluate the efficiency of VSFS, we conduct a representative HiveQL query: "*find the minute in which the*

TrionSort cluster contains the highest number of the high-latency events caused by $Writer_5$", where $Writer_5$'s latency has been recognized as the "interesting feature" [51]:

```
SELECT minute, count(minute) AS mincount
FROM (SELECT round(time / 60) AS minute FROM
    trionsort WHERE attr_name = 'Writer_5_runtime'
and attr_value > 2000000) t2 GROUP BY minute ORDER
    BY mincount DESC LIMIT 1;
```

Code 4. The HiveQL query for finding the 1-minute time period when there are the most "Write_5_runtime" events whose lengths are greater than 2,000,000 ms.

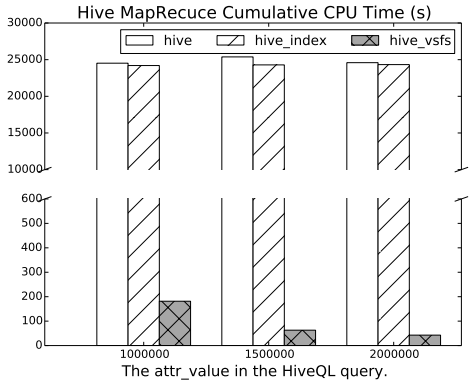

Fig. 7. Hive Speedup On Different System Models (Hive without index, Hive with index, and Hive with VSFS)(10s/file).

First, we issue the HiveQL query in Code 4 to three system models: Hive without index, Hive with index, and Hive with VSFS in which Hive uses VSFS to manage namespace and data-filtering, labeled "hive", "hive_index" and "hive_VSFS" respectively in Figure 7. Then, we choose 1000000, 1500000 and 2000000 as the attr_value thresholds for the query. We measure the cumulative CPU time for the HiveQL MapReduce tasks, instead of the overall execution time (i.e., the wall-clock time), which is time spent on the MapReduce job between when the job is submitted and when it is completed, because it is more accurate for representing the scale of the computation. However, for the Hive-with-VSFS system model, only $6 \sim 23$ mappers are required after the unrelated data have been filtered out, which means that our testing cluster is far from being saturated. As a result, the wall-clock times taken by the Hive MapReduce tasks are similar for all the tests under the Hive-with-VSFS system model. Nevertheless, the overall execution performance as measured by the wall-clock time in the tests under the Hive-with-VSFS system model is $15.4 - 198.2$ times faster than that in the tests under the Hive-without-index or Hive-with-index system model. The results of the 10-second log file dataset are presented for this evaluation, while results from other datasets are similar and thus omitted. As illustrated in Figure 7, using indices in Hive has no significant improvement for the given queries, because Hive ignores the index for large scans. In the Hive-with-VSFS evaluation, the higher the "attr_value" it chooses, the more data can be filtered out, therefore the higher speedup can be achieved. In fact, VSFS is able to speed up Hive by 90.8, 402.8 and 942.7 times, when we use 1000000, 1500000 and 2000000 as the attr_value threshold respectively.

Next, we conduct a sensitivity study to assess the performance impact of the file size in the dataset for VSFS. We use the same dataset with 3 different file sizes (i.e., 1-, 10-, and 30-second records). The same HiveQL query in Code 4 is executed on the three datasets, and the computation time and HDFS IOs are collected. Figure 8 clearly demonstrates that the smaller the file is, the finer grained control the filtering process can achieve, resulting in higher computation speedups (up to 1779×), which is measured by T_{hive_index}/T_{vsfs}. However, finer grained file-index control inevitably comes at the cost of hosting a *larger* number of *smaller* files in the system, as well as higher indexing overhead.

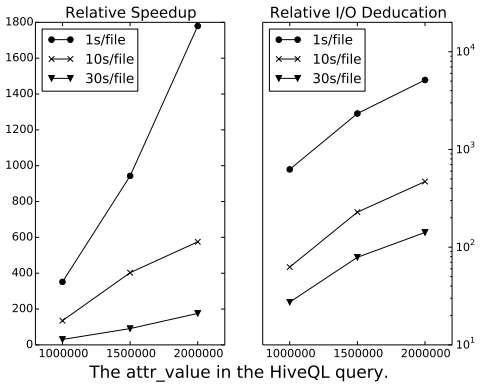

Fig. 8. The Impact of File Size, i.e., Indexing Granularity, on Hive Speedup and I/O Reduction.

In summary, VSFS' file-level indexing and search capabilities can serve to effectively and efficiently filter out data for a class of analytics applications without introducing false negatives.

V. Conclusion and Future Work

This paper presents VSFS, a novel file system to address the needs for data filtering at the filesystem-level for big data analytics applications. By offering near flexible real-time file-indexing and file-search capabilities, VSFS is able to accelerate real world analytics applications significantly. We are working on an HDFS-compatible VSFS to transparently accelerate Hadoop MapReduce framework. We also plan to evaluate VSFS on larger platforms, such as EC2, and further improve its scalability and reliability.

References

[1] Castor: Cern advanced storage manager. http://castor.web.cern.ch.
[2] Filesystem hierarchy standard. http://www.pathname.com/fhs/.
[3] Large Harden Collider. http://lhc.web.cern.ch.
[4] VoltDB. http://voltdb.com.
[5] C. L. Abad, N. Roberts, Y. Lu, and R. H. Campbell. A storage-centric analysis of mapreduce workloads: File popularity, temporal locality and arrival patterns. In *IISWC*, pages 100–109. IEEE Computer Society, 2012.
[6] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. EuroSys '13.
[7] Amazon.com. Amazon simple storage service(s3). http://aws.amazon.com/s3/.
[8] R. Ananthanarayanan and et al. Photon: Fault-tolerant and scalable joining of continuous data streams. SIGMOD '13.
[9] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: a fast array of wimpy nodes. SOSP '09, New York, NY, USA, 2009. ACM.
[10] Apache.org. Apache hadoop. http://hadoop.apache.org/.

[11] Apache.org. Hadoop distributed file system.

[12] Apple Inc. Spotlight. http://www.apple.com/macosx/what-is-macosx/spotlight.html.

[13] K. Banker. *MongoDB in Action*. Manning Publications Co., Greenwich, CT, USA, 2011.

[14] Basho. Riak, an open source, distributed database. http://docs.basho.com/riak/latest/references/appendices/concepts/Replication/.

[15] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18, 1975.

[16] D. Borthakur and et al. Apache hadoop goes realtime at facebook. SIGMOD '11. ACM.

[17] R. Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[18] S. Chaudhuri, G. Das, and V. Narasayya. Optimized stratified sampling for approximate query processing. *ACM Trans. Database Syst.*, 32(2), June 2007.

[19] M. H. Chin and et al. Induced pluripotent stem cells and embryonic stem cells are distinguished by gene expression signatures. *Cell Stem Cell*, 5(1):111 – 123, 2009.

[20] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. NSDI'05.

[21] J. C. Corbett and et al. Spanner: Google's globally-distributed database. In *OSDI '12*.

[22] R. C. Daley and P. G. Neumann. A general-purpose file system for secondary storage. In *AFIPS '65 (Fall, part I): Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, 1965.

[23] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. OSDI'04, Berkeley, CA, USA, 2004. USENIX Association.

[24] G. DeCandia and et al. Dynamo: amazon's highly available key-value store. SOSP '07.

[25] D. Duellman. Cern storage update, 2008.

[26] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole, Jr. Semantic file systems. In *SOSP '91*, 1991.

[27] Google.com. Google Desktop Search. http://desktop.google.com/.

[28] Google.com. Google Search Applicance. http://www.google.com/enterprise/search/gsa.html.

[29] B. Gopal and U. Manber. Integrating content-based access mechanisms with hierarchical file systems. In *OSDI '99*.

[30] R. Hagmann. Reimplementing the cedar file system using logging and group commit. *SIGOPS Oper. Syst. Rev.*, 21(5), Nov. 1987.

[31] Y. He and et. al. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*.

[32] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. SIGMOD '97.

[33] R. Henschel, S. Simms, D. Hancock, S. Michael, T. Johnson, N. Heald, T. William, D. Berry, M. Allen, R. Knepper, M. Davy, M. Link, and C. A. Stewart. Demonstrating lustre over a 100gbps wide area network of 3,500km. SC '12, 2012.

[34] Y. Hua, H. Jiang, Y. Zhu, D. Feng, and L. Tian. Smartstore: a new metadata organization paradigm with semantic-awareness for next-generation file systems. In *SC '09*, 2009.

[35] H. H. Huang, N. Zhang, W. Wang, G. Das, and A. S. Szalay. Just-in-time analytics on large file systems. In *FAST '11*, 2011.

[36] A. K. Jain, L. Hong, and S. Pankanti. To BLOB or not to BLOB: Large object storage in a database or a filesystem? Technical Report MSR-TR-2006-45, Microsoft Research, April 2006.

[37] K. Janowicz. Big data giscience? In *Big Data in Geographhic Information Science Panel 2012*, 2012.

[38] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. STOC '97.

[39] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. WWW '99.

[40] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2), June 1981.

[41] L. L. N. Laboratory. SLURM: A highly scalable resource manager. https://computing.llnl.gov/linux/slurm/slurm.html.

[42] A. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller. Spyglass: Fast, scalable metadata search for large-scale storage systems. In *FAST '09*.

[43] J. Lin and D. Ryaboy. Scaling big data mining infrastructure: the twitter experience. *SIGKDD Explor. Newsl.*, 14(2):6–19, Apr. 2013.

[44] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum. In-situ mapreduce for log processing. USENIX ATC'11.

[45] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. SIGMOD '10.

[46] E. R. Mardis. Next-generation DNA sequencing methods. *Annu. Rev. Genomics Hum. Genet.*, 9:387–402, 2008.

[47] S. Margo and M. Nicholas. Hierarchical file systems are dead. In *HotOS '09*, 2009.

[48] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. VLDB '2010, pages 330–339, 2010.

[49] Microsoft. Windows Search. http://www.microsoft.com/windows/products/winfamily/desktopsearch/default.mspx.

[50] Microsoft. WinFS: Windows Future Storage. http://en.wikipedia.org/wiki/WinFS.

[51] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. NSDI'12.

[52] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. SIGMOD '08.

[53] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. SOSP '11.

[54] S. Patil and G. Gibson. Scale and concurrency of GIGA+: File system directories with millions of files. In *FAST '11*, 2011.

[55] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. SIGMOD '09.

[56] K. Ren and G. Gibson. Tablefs: Enhancing metadata efficiency in the local file system. In *USENIX ATC'13*.

[57] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. Nobody ever got fired for using hadoop on a cluster. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, HotCDP '12, 2012.

[58] M. Stonebraker and U. Cetintemel. "One Size Fits All": An idea whose time has come and gone. In *ICDE '05*, 2005.

[59] C. the European Organization for Nuclear Research. Compact muon solenoid experiment at cern's lhc. http://cms.web.cern.ch/.

[60] R. Thomsen and M. H. Christensen. Moldock: A new technique for high-accuracy molecular docking. *Journal of Medicinal Chemistry*, 49(11):3315–3321, 2006.

[61] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, Aug. 2009.

[62] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at facebook. SIGMOD '10.

[63] D. Tiwari, S. Boboila, S. Vazhkudai, Y. Kim, X. Ma, P. Desnoyers, and Y. Solihin. Active flash: Towards energy-efcient, in-situ data analytics on extreme-scale machines. FAST' 13.

[64] 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 *OSDI '06*, 2006.

[65] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. PDSW '07.

[66] B. Welch and et al. Scalable performance of the panasas parallel file system. In *FAST '08*.

[67] A. Wilson. The new and improved filebench. In *Proceedings of 6th USENIX Conference on File and Storage Technologies*, 2008.

[68] L. Xu, H. Jiang, X. Liu, L. Tian, and J. Hu. Propeller: A scalable metadata organization for a versatile searchable file system. Technical Report 199, University of Nebraska Lincoln, Mar. 2011.

[69] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. HotCloud'10.

[70] F. Zheng and et al. In-situ data analytics and reduction. SC '13.