VSFS: A Searchable Distributed File System

Lei Xu
Cloudera
lei@cloudera.com

Ziling Huang
NetApp
ziling.huang@netapp.com

Hong Jiang, Lei Tian, David Swanson
University of Nebraska-Lincoln
{jiang,tian,dswanson}@cse.unl.edu

***Abstract*—In this paper, we propose a Versatile Searchable File System, *VSFS*, which builds POSIX-compatible namespace using a novel Namespace-based File Query Language (NFQL). This enables analytics applications to utilize VSFS high-performance file-search service without changing their data model. VSFS versatile file-indexing mechanism is designed to offer great flexibility for applications to control indices to satisfy analytics needs. The evaluations driven by two real-world analytics applications demonstrate VSFS' high scalability and powerful data-filtering functionality.**

***Index Terms*—File System Managements, Distributed Systems, Indexing Methods, Information Filtering**

I. Introduction

Large scale distributed and parallel file systems [8], [17], [22], [45], [46] are the *de-facto* data management infrastructures in the big data [29], [44] and scientific computing environments [2], [42]. Compared to the various database solutions, including the traditional SQL databases [3], the NoSQL databases [10], [12], [16] and the NewSQL databases [4], [14], file systems usually shine by providing better scalability (i.e., larger volume and higher parallel I/O performance) and flexibility (i.e., supporting both structured and unstructured, as well as non-fixed data schemas). Therefore, a large fraction of existing analytics applications (e.g., MapReduce [7] and scientific computing applications [2], [41]) are still using file systems to access raw data.

The file system's superior performance largely comes from its data organization that de-couples the hierarchical namespace from the actual data. This de-coupling of data and organization introduces enormous opportunities for the file system designers and application developers to optimize I/O access patterns and to increase parallelism [22], [36], [46].

However, with large volumes of complex datasets, such as system logs, web clicks, financial transaction records and scientific experimental data [2], [44], the decades-old hierarchical filesystem namespace concept [15] is starting to show the impact of aging, falling short of managing such complex datasets in an efficient manner [23], [28], [32], [47]. The root cause of this inefficiency can be traced to the fact that *the file path in the hierarchical namespace is the only identity of data*. First, to ensure the uniqueness of the data identity, the file path must be sufficiently descriptive to be distinguishable among billions of files. As a result, the filesystem hierarchical namespaces on such large systems become complex and difficult to maintain [48]. Second, the efficiency of retrieving the desired data from the file system is restricted by the hierarchical directory schemes [18], [20], [32]. In other words, organizing files (data) in the directory hierarchy can only be effective and efficient for the file lookup requests that are well aligned with the existing hierarchical structures. For today's highly variable can complex big data and scientific datasets, a pre-defined directory structure can hardly foresee, let alone satisfy the *ad-hoc* queries that are likely to emerge.

While there have been extensive studies attempting to address the problem of filesystem management inefficiency by offering file search functionalities from desktop to enterprise systems [18]–[20], [23], [24], [27], [28], [33], few such systems are designed for data intensive environments where the applications desire the file systems to return consistent file search results in real-time or near real-time. For such environments a scalable and high-performance indexing facility becomes a necessity, which in and by itself is a challenging design goal. Additionally, most of these studies are confined to the scope of file search engines or advanced metadata management, which can not overcome the essential limitations of the current form of hierarchical file systems in the big data environments, in which the datasets are often characterized as 3"V"s: *Volume, Velocity and Variety*.

In this paper, we rethink the notion of file system namespace and how applications interact with the file systems, by envisaging file search to be one of the primary interfaces, on a par with the POSIX interface, to organize and retrieve data. To realize this vision, we propose a new form of file system, called *Versatile Searchable File System* (**VSFS**), that is designed as a large-scale distributed filesystem that *dynamically builds namespace based on the file-search API* and *offers a new computational model*, i.e., computing through file-search, to interact with analytics applications. A range of analytics applications [2], [13], [31], [35], [41], [42] can benefit from VSFS file-search capability to accelerate computing by leveraging its the file-search service to achieve coarse-grained (file-level) data filtering to dramatically reduce the scale of the input data, resulting in significantly reduced computation time.

To this end, VSFS offers the following salient features that differentiate it from the existing solutions:

- *Namespace-based file query language*. VSFS builds namespace using a novel Namespace-based File Query Language (NFQL) that is compatible with the POSIX hierarchical namespace [1]. VSFS answers the query as a dynamically generated directory and fills it on-demand with the symbolic links to the desired files [18]. Thus, this backward capability allows the existing analytics

applications to run on VSFS and to adopt the latter's advanced file-search functionalities without *code modification*. More importantly, it enables a new flexible way for applications to organize and retrieve data.
- *Versatile and near real-time file indexing*. Being the key enabler of VSFS and NFQL, a flexible and high-performance indexing mechanism is carefully designed to create and customize file indices anywhere within the file system on the fly. Its near real-time file indexing enables highly-accurate file search, because all relevant data is updated and none is left un-indexed.

II. Motivation: Data Filtering via File Search

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

Due to the huge volumes of big data datasets and strong needs for inspecting different aspects of the same datasets, the ability to flexibly filter data has become increasingly critical for the analytics applications to accelerate the computation [2], [5], [49], [50]. We present three file-based filtering methods with their corresponding suitable applications and datasets to give the readers an idea of how the proposed file-search interface can be widely applicable in different scenarios (Table I).

Data filtering method	Applicable scenario
Accurate file filtering	Each input file is an individual data unit and filtering returns the exact set of qualified files. For instance, Melegro Virtual Docker [42].
Approximate file filtering with possible false negatives	Probabilistic and statistical results with a small probability of useful data being filtered out, e.g., machine learning (Distalyzer [35]), social data mining.
Approximate file filtering without false negative	Requires pre-scan to filter out unwanted data (e.g., OLAP). For instance, Hive [43].

TABLE I
DATA FILTERING METHODS AND THEIR APPLICABLE SCENARIOS.

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. We broadly divide them into the following three categories:

File search engines [9], [19], [33], which rely on the crawling process to catch up with new updates periodically, are unlikely to keep the file index always up-to-date [28], [48]. This can lead to inaccurate retrieval results. Thus, many data analytics applications cannot 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 [2], [30], [50].

Database-based metadata services use databases as a supplementary file metadata management service running on top of file systems [41]. These database-based metadata service share the same limitations of database-based storage solutions [10], [12], the performance of which could not match the I/O workloads on file systems [22], [28], [46]. Additionally, the static and stable SQL schema is not well suitable for the exploratory and ad-hoc nature of many big data and HPC analytics activities [29], [37]. Finally, the separation between the file system and the file index will easily lead to an inconsistent state. It is a well-accepted fact that databases are not a "one-size-fits-all" solution [40].

Searchable file system interfaces provide file search functions directly through the file systems. Research prototypes that attempt to provide such interfaces include Semantic File System [18], HAC [20] and WinFS [34]. 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 [23], [28]. These queries may not be meaningful for many 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 limits the flexibility in handling the very high variety and heterogeneity of the datasets.

III. Design and Implementation

VSFS' two key features, 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 filesystem-level real-time file indexing and search capability.

A. NFQL 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 [15] so that *legacy applications are able to run and search on VSFS without any modifications.*

NFQL. VSFS inherits the concept of *using the POSIX directory semantics to perform file search* [18], [20], [34]. In addition, because VSFS' NFQL is purposely designed to support analytics applications, its semantics is more flexible and richer than the content-based queries in the existing schemes. 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) being located under the directory "*/foo/bar*" (including its subdirectories); 2) energy targeted at "drug-A" being greater than* 10.5 *eV; and 3) weights being smaller than* 16 *kilodaltons*" by simply scanning the following directory in NFQL:

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

A simplified NFQL specification is listed in Grammar 1. In an NFQL query, the *prefix* directory must be a physically existing directory. The multi-dimensional query uses a form similar to a zero-based array to access a particular dimension of one indexed attribute (e.g., "$coord[2] > 10$"). 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 a 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 of its dynamical generation, this virtual directory is only visible to the client who issues the query, where the original dataset is not changed [49] and no data movement is incurred [29].

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 a pre-determined index schema (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. 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.

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.

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").

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 (i.e., B-tree, hash table or K-D-Tree [11]) for this file index.

Finally, *key type* describes the data type used for the key of an index (e.g., int or string). It is important to provide such choices for the users because in big data environment there are various demands to store different key types.

With such 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 [9], [18], [34]. 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. These design choices significantly differentiate VSFS from the existing systems [18], [23], [28].

B. RAM-based Distributed Index

We design a scalable RAM-based distributed index architecture that enables VSFS to offer real-time file-indexing and file-search capabilities. To simplify development, the raw file data in VSFS is managed by the existing matured storage systems and abstract them to an ObjectStorage interface for VSFS to use [38], [45]. In its entirety, a metadata and index cluster of VSFS consists of two Consistent Hashing (C.H) Rings [16], where one C.H ring is constructed by *Master Servers* and the other by *Index Servers*. Additionally, analytics applications can use either VSFS' API or the FUSE-based VSFS client to access VSFS' file-filtering service and the object store, as shown in Figure 1.

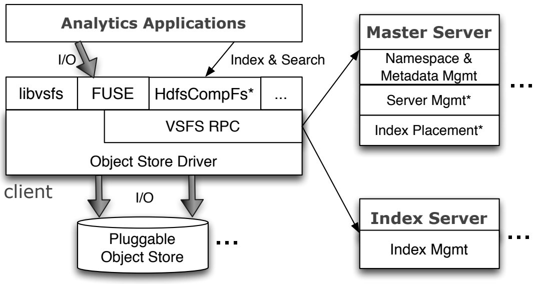

Fig. 1. The VSFS Architecture. (*)HdfsCompFs is an HDFS compatible file system under development. (*)Server management and index placement are only managed by the primary master server.

Master Server takes charge of the namespace for both files and indices, as well as file metadata. Namespace and metadata are distributed to the master server cluster's 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 the topology information of all master servers and index servers. The cluster topology information of this cluster will be managed by ZooKeeper [25] in our future work.

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. VSFS must efficiently map between the file path and oid in both directions for fast resolving the file paths from the oids queried from the index servers. Hence, VSFS uses its own master servers to take over the responsibility of filesystem namespace management. Furthermore, 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, and the remaining 48 bits are assigned by a server-wide unique value. Hence, it ensures that the oid has the *exact* same distribution as the hashed value of the file path. Using "oid" instead of the actual file path in an Index Server also makes it easier to rename or delete files. Thus, VSFS is able to return consistent file-search results with such namespace changes.

An **Index Server** manages various kinds of file indices and answers client's index/search requests. To achieve the lowest-possible file-indexing and -search latencies, each index server keeps all file indices within its RAM, and uses write-ahead-logs [21] to offer the durability of a file index.

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 an index key in the form of a string and lets the index server interpret the key.

Furthermore, to scale and balance large file indices, a large index is divided into smaller *Index Partitions*, managed by a logical C.H ring. If the scale of a partition exceeds a certain threshold, it will be divided and live migrated to one of the other index servers [6]. This per-index logical C.H ring is also managed by the primary master server. As a result, VSFS can statistically balance index partitions globally.

Currently, VSFS provides a basic level of failure-tolerance capability in case of 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 the 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.

VSFS Client parses the requests from the analytics applications through VSFS' API or the FUSE implementation, creates a query plan, and issues the requests through RPCs to the VSFS cluster. In addition, both the master server C.H ring and the index server C.H ring are aggressively cached in the FUSE daemon. When processing file-indexing or search requests, the VSFS client interacts with multiple index servers in parallel to reduce the latency. It also manages to fill the virtual directory in FUSE with the symbolic links to the resultant files for file-search requests.

IV. Preliminary Results

Experimental Setup. We prototype VSFS on a 20-node heterogeneous Scientific Linux 6.3 cluster. Each node features 1 ∼ 2-socket AMD Opteron CPU with 8GB RAM and 60GB local disk to store experimental data. 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 10Gb Fiber channel connection.

A. Indexing Performance

We compare VSFS's indexing performance against the MySQL Cluster 7.2.10, HBase 0.94.6, MongoDB 2.5.1 and VoltDB 3.4, because they represent the state-of-the-art file management solutions in large-scale systems. All systems are optimized to the best of our knowledge. 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. Additionally, we choose 1024 requests as the batch size for all tests. In each test, the clients issue a total of 10 million records.

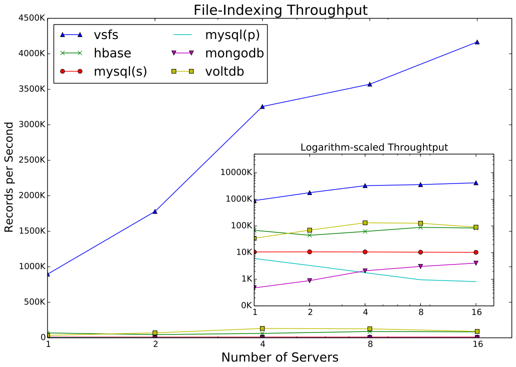

Fig. 2. File-Indexing Throughput.

As illustrated in Figure 2, 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 (MySQL(s)) is used and $1492 \sim 4796\times$ faster when the partitioned SQL tables (MySQL(p)) 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. Its throughput even drops after the cluster scale 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 significantly outperforms the existing solutions in file-indexing performance.

B. Application Performance

In this subsection, we run two real-world applications, namely, Melegro Virtual Docker (MVD) [42] and Hive [43], to demonstrate VSFS' data filtering capability. 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. We configure a 4-node sub-cluster to run the file-search service (1 master server and 3 index servers) and use the remaining 16 nodes as worker nodes for the applications of MVD and Hive.

Molegro Virtual Docker (MVD) [42] represents the category of applications [2], [26] that can utilize the *accurate filtering* strategy (Table I). 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 comparison (existing) solutions.

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 production Lustre file system and generates one record ("(key, file)" pair) to be indexed.

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. In the original MVD case (i.e., denoted by "*MVD*" in Table II), which represents its current practice, 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 II, the execution time of the MVD analytics can be significantly reduced when using the external file-search service. However, 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.

Hive [43]. We use Hive, a data warehouse system on Hadoop, to demonstrate how to utilize VSFS to *filter data without false negatives* (Table I) to accelerate computing [29], [44].

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 II
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.

The TrionSort dataset from Distalyzer [35] is used and intensified to achieve a 200GB-scale dataset [39] by being replicated 300 times with the timestamp of each record unchanged. The intensified dataset is placed in an HDFS directory without being re-organized, 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. All columns that will be used in the HiveQL query conditions are built with Hive indices.

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" [35]:

```
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 1. 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.

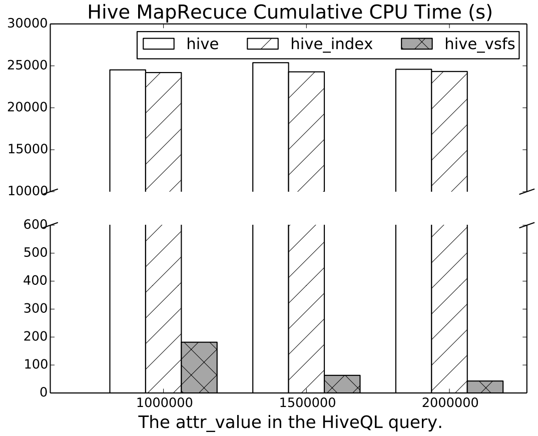

Fig. 3. Hive Speedup On Different System Models (Hive without index, Hive with index, and Hive with VSFS).

First, we issue the HiveQL query in Code 1 to three system models: Hive without index, Hive with index, and Hive with

VSFS in which Hive uses VSFS to manage namespace, labeled "hive", "hive_index" and "hive_VSFS" respectively in Figure 3. Then, we choose $1000000, 1500000$ and 2000000 as the attr_value thresholds for the query. The cumulative CPU time for the HiveQL MapReduce tasks is measured. As illustrated in Figure 3, using indices in Hive provides 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.

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 and HPC analytics applications. By offering near flexible real-time file-search capabilities, VSFS is able to accelerate real world analytics applications significantly. We plan to evaluate VSFS on larger platforms, such as EC2, and further improve its scalability and reliability.

References

[1] Filesystem hierarchy standard. http://www.pathname.com/fhs/.

[2] Large Harden Collider. http://lhc.web.cern.ch.

[3] Oracle database. http://www.oracle.com/us/products/database/overview/index.html.

[4] VoltDB. http://voltdb.com.

[5] 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.

[6] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: a fast array of wimpy nodes. SOSP '09.

[7] Apache.org. Apache hadoop. http://hadoop.apache.org/.

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

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

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

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

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

[13] M. H. Chin and et al. Induced pluripotent stem cells and embryonic stem cells are distinguished by gene expression signatures. *Cell Stem Cell*, 2009.

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

[15] R. C. Daley and P. G. Neumann. A general-purpose file system for secondary storage. In *AFIPS '65*.

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

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, (5), 2003.

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

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

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

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

[22] R. Henschel and et al. Demonstrating lustre over a 100gbps wide area network of 3,500km. SC '12.

[23] Y. Hua and et al. Smartstore: a new metadata organization paradigm with semantic-awareness for next-generation file systems. In *SC '09*.

[24] 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.

[25] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. USENIXATC'10, 2010.

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

[27] C. Johnson, K. Keeton, C. B. M. III, C. A. N. Soules, A. Veitch, S. Bacon, O. Batuner, M. Condotta, H. Coutinho, P. J. Doyle, R. Eichelberger, H. Kiehl, G. Magalhaes, J. McEvoy, P. Nagarajan, P. Osborne, J. Souza, A. Sparkes, M. Spitzer, S. Tandel, L. Thomas, and S. Zangaro. From research to practice: Experiences engineering a production metadata database for a scale out file system. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 191–198, Santa Clara, CA, 2014. USENIX.

[28] 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*.

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

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

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

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

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

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

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

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

[37] 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.

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

[39] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. Nobody ever got fired for using hadoop on a cluster. HotCDP '12.

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

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

[42] R. Thomsen and M. H. Christensen. Moldock: A new technique for high-accuracy molecular docking. *Journal of Medicinal Chemistry*, 2006.

[43] A. Thusoo and et al. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2009.

[44] 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.

[45] 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.

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

[47] L. Xu. *Scalable file systems and operating systems support for big data applications*. PhD dissertation, University of Nebraska Lincoln, 2014.

[48] L. Xu, H. Jiang, L. Tian, and Z. Huang. Propeller: A scalable real-time file-search service in distributed systems. ICDCS '14.

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

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