Spring 4-20-2013

VSFS: A Versatile Searchable File System for HPC Analytics

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

Ziling Huang
University of Nebrask, 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

VSFS: A Versatile Searchable File System for HPC Analytics

Lei Xu, Ziling Huang, Hong Jiang, Lei Tian
Department of Computer Science & 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

Emerging HPC analytics applications urgently demand file-search services to drastically reduce the scale of the input data in real-time, so that the speed of computation and data analytics can be greatly accelerated. Unfortunately, the existing file-search solutions are either poorly scalable for large-scale systems, or lack a well-integrated interface to allow applications to easily use them for critical tasks. We believe that the time is ripe for the design of a searchable file system capable of accurate and scalable system-level file-search functionality.

In this paper, we propose a Versatile Searchable File System, *VSFS*, which provides a transparent, accurate and real-time file-search service through a POSIX-compatible file system namespace that can be integrated into any HPC/Big Data legacy code without modifications. Additionally, to support real-time file search, VSFS uses a DRAM-based distributed architecture to perform real-time file indexing. Moreover, a versatile index scheme is designed to adapt to the various forms of HPC datasets. The results of our VSFS prototype evaluation show that VSFS is scalable in a typical HPC environment. It achieves significantly better file-indexing and file-search performance than the popular SQL/NoSQL solutions, while it only introduces negligible I/O overhead. Finally, we integrate VSFS to a scientific analytics application to show its benefits in terms of performance and ease of use.

1. INTRODUCTION

For historical reasons, the HPC community is still largely relying on traditional file systems [38, 62, 63] to store and manipulate scientific data for HPC applications. However, traditional file systems, which are built based on the notion of a hierarchical namespace, are well-known for their lack of data manageability and efficient retrievability [39, 46, 50]. Thus, as illustrated in Figure 1.(a), many HPC applications have to integrate the data management logic into the application [8, 30, 60]. This places an unnecessary and often times heavy burden on the domain experts who code their applications. As we enter the Big Data era, this model of developing applications is no longer suitable because Big Data's defining characteristics of *Volume, Velocity*, and *Variety*, have further exacerbated the difficulty of large-scale scientific data management, which far exceeds the capability and responsibility of a single HPC analytics application or its developers.

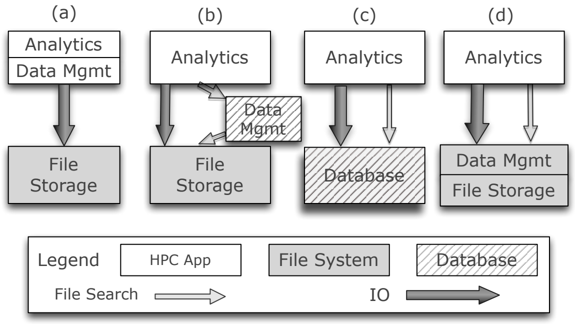

Figure 1: HPC Data Management Approaches: a) Application takes the responsibly of managing data, b) Additional data management service runs on top of file system, c) Use SQL/NoSQL databases as data store, d) File system provides data management facility.

One alternative to this model is to build a data management service (e.g., using database) that is placed between the application and file system [34, 46, 53], as shown in Figure 1.(b). This alternative model shifts the data management responsibility from applications into the data management service. For example, CERN uses Oracle DB [6] and MongoDB [18] to manage the file metadata and store the raw data on the Lustre file system [30]. Another alternative model is to store the entire dataset into SQL or NoSQL databases (Figure 1.(c)). This model has long been adopted by enterprises, such as BigTable [23], Dynamo [29], RamCloud [55], VoltDB [10], Spanner [26], to satisfy the requirements of enterprise Big Data analytics. However, very few of them have been actually deployed in the HPC (Scientific Computing) environments because they are not able to store the huge number of unstructured files that are typically generated by and used in HPC applications [41]. More importantly, both of the aforementioned alternative data-management approaches suffer from two common problems:

- *POSIX Incompatibility.* Most of the existing scientific applications require a POSIX file system API to access data. Using SQL/NoSQL databases require domain experts (e.g., biologists, physicists, etc.) to learn the particular POSIX-incompatible APIs (e.g., SQL language) and data models, as well as to re-write a large portion of the existing HPC applications, which would is extremely expensive, even infeasible.

- *Insufficient Throughput and Scalability.* HPC applications [5,30] demand extremely high throughput and parallel accesses, which is often far beyond the capability and scalability of the SQL/NoSQL-based solutions.

Thus, we argue that it is high time that a new storage system model be proposed for HPC analytics applications in today's Big Data environment. This model should offer data-management functions directly from a POSIX-compatible file system API (Figure 1.(d)), so that it completely frees the legacy HPC applications and domain experts from data management. To this end, we propose a *Versatile Searchable File System (VSFS)* to bridge the gap between traditional file systems and the need of HPC and Big Data applications for flexibility, ease of programmabling and powerful data management. VSFS provides a *transparent POSIX-compatible searchable file system namespace* that allows users to search their desired files directly through POSIX directory semantics [33]. Additionally, to preserve the consistency required by file system semantics, VSFS provides real-time and accurate file-search results, which are enabled by VSFS' real-time file-indexing capability. Finally, a *versatile* index scheme is designed to adapt to the various forms of HPC/Big Data datasets. We discuss the detailed file system namespace and API design in Section 3.1.

To this end, the paper aims to make the following main technical contributions:

1. The introduction of a new file system model for HPC/Big Data analytics, i.e., a versatile searchable file system, and the proof of its necessity and feasibility [36, 39, 40, 46, 50, 58]. A comprehensive discussion is provided on the design principles of such a file system and its possible interfaces.

2. The development of a distributed DRAM-based searchable file system prototype with a novel file system namespace and file-index scheme, VSFS, that has *very low indexing overhead* so that it can be realistically deployed in an HPC environment, and returns *file-search results* in real-time so that the requirements of the POSIX file system semantics are met.

3. Extensive evaluation demonstrating VSFS' feasibility of being deployed in data-intensive environments. VSFS significantly outperforms a number of existing and state-of-the-art file-search solutions (i.e., a clustered SQL database (MySQL cluster), NoSQL database (HBase)) in file-indexing (up to 4709 times) and file-search (up to 6275 times) performance at a reasonable and acceptable I/O overhead. VSFS reduces the computation time of a HPC analytics application (i.e., Molegro Virtual Docker [60]) dramatically without modifying a single line of the application code.

The rest of this paper is organized as follows. Section 2 presents the necessary background to motivate the research on VSFS. The design and implementation of VSFS, along with the system-level search API, are described in Section 3. We evaluate the scalability, performance and effectiveness of the VSFS prototype in Section 4. Section 5 concludes the paper with remarks on directions of future research on VSFS.

2. RELATED WORK, BACKGROUND AND MOTIVATION

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

2.1 The Need of File-Search Capability for Big Data Analytics in HPC

HPC and Big Data analytics applications tend to store scientific data in files rather than in databases [5,24,30,47–49,60] for the reasons of 1) compatibility with legacy code, 2) better performance and higher scalability provided by parallel file systems [38,56,63], 3) convenience of sharing data with and distributing data to colleagues, and 4) less storage/database knowledge required of domain experts to manipulate data.

Unfortunately, file systems have their own inherent weaknesses in slow extracting meaningful information that stems from *the singular and static file representation (i.e., file path) and results in inefficient data management and retrieval* [39, 46,50]. Clearly, an efficient file-search solution to effortlessly organize and manage scientific data is desirable and critically important in the Big Data era.

Here we present a typical HPC analytics application as a running example throughout this paper to illustrate the needs for, key ideas pertaining to, and evaluation of, an alternative and novel approach to Big Data storage and management in scientific discovery:

- *Computational Drug Discovery.* Molegro Virtual Docker (MVD) [60] is a proprietary software that is widely used in both industry and academia to predict and analyze protein-ligand interactions. In the current practice, for each run, MVD brute-forcedly runs 1 ligand compound (e.g., out of a total of 1000 ligand compounds) against all protein targets (e.g., 10-million protein targets) that are stored in separate plain text files (i.e., 10 million files). For each pair of protein and ligand, one output file is generated independently during the MVD computation. Then, the scientists need to compare each set of output files for every target to find the commonality and uniqueness. Moreover, the scientists desire to be able to easily gather a particular sub-set of the protein-target files for each run to shrink the amount of objects to compute. The criteria used to search the sub-set of input files can be pre-defined (e.g., family, race, gender) or be calculated from the previous runs (e.g., the protein docking results). Since the computational complexity is $O(mn)$, where m is the number of ligand compunds and n is the number of protein targets, it is obviously desirable to have the capability of dynamically and selectively filtering the

Data Mgmt Solution Model	Manageability	Scalability	Performance	POSIX-API	Examples	Notes
(a) Application Data Mgmt	Weak	Poor	Good	Yes	MVD [60], Stem Cell Research [24]	Hard to implement, customizable
(b) Supplementary Data Mgmt Service	Fair	Poor	Poor	NO	Spyglass [46], SmartStore [39], CASTOR [1], Google Enterprise Search [34]	Poor programmability, Inaccurate for crawler-based solutions [65], Limited Indexable Attributes [15,34,39,46]
(c) SQL/NoSQL Database as data storage	Good	Poor(RDBMS)/Good(NoSQL)	Fair	No	MySQL, OracleDB, HBase, MongoDB	No large blobs, Database knowledge required
(d) Filesystem-level Data Mgmt	Good	Good	Good	Yes	VSFS	Reserve filesystem's advantages (e.g., large blobs, high throughput).

Table 1: Comparison of HPC/Big Data Management Solutions.

input dataset, so that the overall computation time can be greatly reduced.

In the HPC world, there are many analytics programs that share the same characteristics of desiring a very small, useful "working data set" from a much larger input data set, as in the aforementioned case. For instance, the Large Harden Collider(LHC) project [5], Next-Generation Sequencing [49], Stem Cell Research [24] and Geographic Information Science [42] all use massive amounts of files as their input data repository and all need to find tiny subsets of the files that satisfy certain criteria defined by the domain scientists for their particular instances of solutions. A few of these scientific projects have attempted to implement *ad-hoc* data management solutions, such as the CERN Advanced Storage Manager (CASTOR) [1,8] for the LHC project. However, such *ad-hoc* solutions not only require significant amount of storage/database knowledge and enormous investments from every research institute, but are also difficult to be expanded to benefit HPC applications from other research areas. Given the clear need for general-purpose file-search capability, there have been many efforts invested by industry and academia to provide alternative solutions for data management that supports file search. In the next subsection, we will examine and compare these solutions to point out their problems and motivate our VSFS research.

2.2 Comparisons among Existing Data Management Solutions

Since different Big Data Analytics solutions have different use cases and there is no one-size-fits-all solution [59], it is desirable to identify and understand the use cases before choosing a solution. We provide a key-feature based comparison of current popular data management solutions for HPC and Big Data analytics applications in Table 1.

For the HPC community, although some applications are now attempting to adopt the Enterprise Big Data analytics solutions like Hadoop [13], Massive Parallel Processing (MPP) Database [4,9] and NoSQL [18,21], most of the scientific applications are still relying on traditional parallel file systems [38,63]. The main reasons behind this are:

First, most HPC applications are proprietary or contain substantial amounts of legacy code, making it difficult and expensive to adapt to the new interfaces of existing Big Data management solutions.

Second, traditional RDBMS systems are considered hard to scale [17,26,39,46], while their ACID transactional property limits their deliverable I/O performance [22]. Additionally, traditional RDBMS systems are not suitable for storing large data chunks by nature [41].

Third, new NoSQL solutions usually have non-interoperable data models that are tightly coupled with the particular NoSQL products. Moreover, due to the uncertain lifetime of NoSQL solutions, it may be considered risky to invest heavily in such storage techniques for the sake of long-term data accessibility.

In contrast, the POSIX file system API, along with the hierarchical namespace [2], is well standardized and accepted by all the major operating systems, especially Linux, which is the dominant OS used in HPC environments. Therefore, arguably, data can be stored within file systems for decades without having to worry about the lack of appropriate tools/APIs to access them. It is also hassle-free to migrate data from one file system to another. This data independence brought about by the long-lasting standardized file system API is especially important to the Scientific Computing (HPC) community and the Big Data community, because:

On one hand, the HPC and Big Data communities have long played a pioneering role in developing new and revolutionary techniques, largely driven by their applications' insatiable demands for faster, more efficient, and more scalable computing capabilities. On the other hand, it is equally important for these two communities to guarantee that their data repositories are durable and accessible for long periods of time, as in the data lie their core values.

Moreover, file systems are considered extremely good at providing high throughput, low latency, high volume data storage, highly parallel access, and supporting large files [38, 41,56,63]. Besides, many access behaviors, such as random access (e.g., reading a 1KB record from a large compressed 1GB data [8]) can only be efficiently achieved through the file system API.

Taking all these factors into account, we can safely draw the conclusion that file systems will continue to be the major storage solution for a large portion of HPC applications.

2.3 File-System Support for (Near) Real-Time Big Data Management

To address the inefficient data management issue of file systems, researchers from academia and industry have proposed a number of file-search solutions [34–36,39,40,46]. Although these solutions work well in their intended application scenarios, they do not fit well in the HPC and Big Data analytics workflow:

Big Data applications are largely defined by their high *Volume*, high *Velocity* and high *Variety* and are characterized by their need for (near-) real-time analytics so that useful and valuable insight/knowledge can be extracted from large volumes of data in a timely manner [16,32,51,52,61]. However, neither the crawler-based solutions (i.e., Solution Model (b) of Table 1) [15,35,46], or the current RDBMS-based solutions [53] and the NoSQL-based solutions (i.e., Solution Model (c) of Table 1) [1,18] are able to satisfy this crucial (near-) real-time requirement of big data analytics because of their poor scalability [39,46], relatively low throughput [22,25], and/or the inaccuracy caused by the crawling delay [65]. Additionally, these solutions [15,34,39,46,54] are inflexible to support such highly variable dataset.

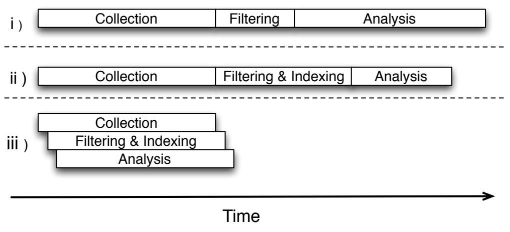

Figure 2: Data processing flow in Big Data analytics. i) No indexing, the traditional approach (i.e., Solution Model (a) of Table 1); ii) Offline indexing, the analysis process is accelerated (i.e., Solution Models (b) & (c) of Table 1); iii) Inline indexing, the entire processing flow can be pipelined (i.e., Solution Model (d) of Table 1).

As a result, we propose an alternative and more suitable approach that provides an advanced, (near-) real-time data management capability directly in the file system. In other words, instead of building file-search add-ons on top of the file system as in Solution Model (b) of Table 1), we design a POSIX-compatible *searchable* file system, VSFS, from the ground-up, which is optimized for real-time file-indexing and file-search operations. With this design methodology, we argue that the file system can preserve the advantages of the existing file system technologies [38,56,62,63] while providing highly optimized file-search solutions by exploring the file system semantics, which is not possible for the RDBMS and NoSQL database based file-search solutions.

This real-time file-search capability in file systems can bring significant advantages to the Big Data analytics and the domain experts. First, as illustrated in Figure 2, with the real-time search capability, it is feasible for Big Data applications to continuously analyze the streamlined input in real time without having to wait for the completion of data collection and indexing (Figure 2.b). This dramatically reduces the window (latency) between collecting raw data and generating meaningful insights (Figure 2.c). Second, providing a POSIX-compatible file-search API directly through the file system can also free domain experts from the laborious task of mastering the prerequisites of processing scientific data, such as SQL syntax, NoSQL data models or the MapReduce framework [28]. Additionally, the consistency of file-search results, which is offered by the real-time file-indexing, allows HPC analytics applications to retrieve data with confidence, which can not be guaranteed by the crawler-based solutions [46] because of inaccurate and outdated results [65].

In summary, to fill the gap between the existing solutions (i.e., Model (b) – add-ons to file systems or Model (c) – SQL/NoSQL databases) and the ever increasing needs of (near-) real-time Big Data analytics, it is necessary to combine the strengths of file systems with databases' analytical capability by offering a (near-) real-time file-search functionality directly from the file system.

3. DESIGN AND IMPLEMENTATION

In this section, we present the design principles and implementation details behind the VSFS system. We first present a detailed description of the design for VSFS's core concept: the transparent searchable namespace and the versatile real-time file indexing, with several examples to illustrate their usage in Section 3.1. Then we describe the DRAM-based distributed architecture that enables the file system level real-time file indexing and search capability in Section 3.2, along with the implementation details of several key components in Section 3.3.

3.1 Searchable Namespace and API

The most unique feature that differentiates VSFS from the other file systems is its flexible and transparent searchable namespace, which is deliberately designed to be backward-compatible with the existing POSIX file systems [27,38,62,63] so that *the legacy codes are able to use VSFS without any modification.* We describe the main VSFS design principles from three aspects: *transparent searchable namespace, versatile file indexing* and *supportive libraries and command-line tool*, as follows.

Transparent Searchable Namespace. VSFS provides file-search capability directly and transparently through its intuitive searchable namespace – it allows the user to use the POSIX directory semantics to perform file-search. In other words, a query is represented by a *virtual path* [33] that contains multiple file-search conditions, for instance, a MVD user can search "*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* 1.6 *kilodaltons*" by scanning the following directory:

"/foo/bar/?drug-A:energy>10.5&weight<1.6/"

VSFS recognizes this directory as a file query and dynamically fills it on-demand with the symbolic links pointing to the actual files that satisfy the specified conditions. Thus, in the above example, by scanning this directory using the POSIX system call "readdir()", or using the standard UNIX command-line tool "ls", users are able to directly obtain the desired results.

Versatile File-Indexing. VSFS aims to provide the flexibility of organizing and managing scientific datasets so that the best file-search performance and experiences can be achieved while requiring of the domain scientists of only minimal stor-

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 2: A file-index definition example for providing range query on float "energy" values.

age system knowledge. It is capable of creating an arbitrary file index on any directory, so that any files under its sub-namespace can be indexed into this file index. A file index is defined by a tuple ***(root, name, index type, key type)***, as detailed in Table 2, and it is uniquely identified by the 2-tuple of ***(root, name)***, where "*root*" is the path of the top directory of the namespace covered by this file-index and "*name*" is a descriptive string of the index. Therefore, domain experts can associate any interesting attribute to their files, which are not restricted to file metadata [39, 46] or to the limited pre-supported attributes [15, 34]. Additionally, as a general file system that supports different HPC/Big Data applications, it is very important to offer the *versatility* to customize file indices. Hence, VSFS currently provides two parameters to customize an index, namely, "*index type*" and "*key type*", as described below:

First, *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 search, "*point*" (Hash Table) for point search and "*muitidim*" (K-D-Tree [20]) for multi-dimensional range search. The modular design of VSFS enables it to be straightforwardly extended to support more index types, for instance, "graph" for graph-based index.

Second, *key type* describes the data type used as the key of the index. VSFS supports the data types of integer, floating-point number, as well as string, as the index key. It is important to provide such choices for domain experts because in the HPC environment there are various demands to store different key types, for instance, double-precision floating-point number (double) for high precision requirements or string for descriptive tagging of input data.

The aforementioned versatile file-indexing mechanism offers great flexibility of management of scientific data. For instance, it allows users to describe the same dataset with any number of attributes by creating multiple indices under the same directory, where each index represents one attribute of the dataset. Additionally, it also allows the users to define the same attribute on different datasets by creating the indices with the same name on different directories. As a use case of VSFS, Melegro Virtual Dock (MVD) [60] is able to build one index (e.g., one with the name "**drug-A:energy**") for each run based on the protein-ligand docking results, so that the inputs of the following MVD analytics can be accurately refined by filtering out the undesired data using this index. It becomes clear that VSFS can dramatically reduce the computational resources required by MVD, among many other HPC applications. In other words, deploying VSFS in HPC environments potentially leads to *faster scientific discovery and better resource utilization*, and ultimately, *freedom of the domain experts from the responsibility of data management.*

```
struct IndexDesc {
  string name;
  int index_type, key_type;
};
struct ComplexQuery {
  struct Range {
    string index_name;
    float low_bound, upper_bound;
  };
  string root;
  vector<Range> ranges;
  ...
};
// Creates an file index on the directory
// 'root' with customized parameters
// in 'desc'.
int create_index(const string &root, const
     IndexDesc& desc);
// Insert a new pair of (file, key) into
// the index.
int update_index(const string &index_name,
     const string &file_path, float key);
// Search files with the conditions in
// 'query'.
int search(const ComplexQuery &query, list
    <string>* results);
```

Example 1: Pseudo-code for VSFS's C++ API

Libraries And Command Line Tool. Additionally, VSFS also provides a command-line tool (vsfsutil) as well as C++/Python/Java libraries. As a result, it offers users the convenience of directly manipulating indices from the command line, as well as an advanced and programmable API for programs to deeply integrate the file-search service. An example of the VSFS C++ API is shown in Example 1.

Finally, we use an example (Example 2) to illustrate how effortless it is to integrate VSFS into an existing application (i.e., MVD) running on a popular HPC resource management system (SLURM [45]):

```
#!/bin/sh
#SBATCH --nodes=100
#SBATCH --time=12:00:00
# Create an index on directory ''/data''
# to support range query on the
# experimental results.
vsfsutil index --create /data --name ''
    drug-A:energy'' --index range --key
    float
# MVD's first run populates the index
# with experimental records through a
# UNIX pipeline.
srun mvd /data | vsfsutil index --name
    drug-A:energy
# MVD's second run takes the refined
# inputs by searching the files with
# the previous records.
srun mvd ''/data/?drug-A:energy>10.5&
    weight<1.6/*''
```

Example 2: SLRUM Job Using VSFS

3.2 VSFS Distributed Architecture

In this subsection, we describe a concrete distributed design that brings the searchable file system concept (Section 3.1) into date-intensive HPC environments.

In order to achieve real-time file indexing and file searching, we build a DRAM-based distributed metadata and index architecture for VSFS. Additionally, we choose to let VSFS treat the raw data storage as a pluggable component that adapts to different types of storage systems (e.g., local file system or networked file systems), so that our current work can focus on the index management. Finally, VSFS, as a prototype to prove the concept of a large-scale searchable file system, aims to scale to hundreds of servers to support tens of Terabytes of file indices and tens of Petabytes of raw data.

VSFS Components. At the architecture level, a VSFS cluster consists of a single *Master Server*, multiple *Index Servers*, multiple *Metadata Servers* and clients, as shown in Figure 3:

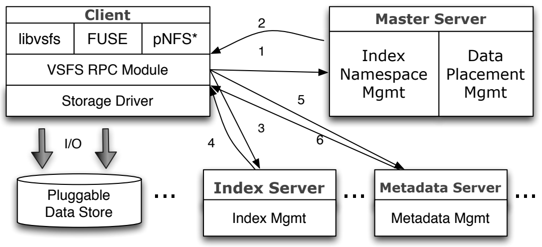

Figure 3: The VSFS Architecture.

- *Master Server* takes charge of the following five functions: 1) it locates the desired index servers for the clients; 2) it maintains the consistent hashing rings for index servers and metadata servers; 3) it assigns the sequence number for each request to provide a global order from the client's point of view; 4) it maintains a mapping between files and indices; and 5) it monitors and manages the status of the entire cluster.
- *Index Server* manages the various file indices and answers the client index/search requests. Each index server keeps all file indices it manages in RAM and takes charge of file indices' durability, recovery and load balancing, which will be elaborated in Section 3.3.
- *Metadata Server* manages the file metadata. In the current prototyping phase, it only manages a mapping from the hash of file path to the string representation of the file path.
- *Client* is in charge of parsing and managing the client-side requests and issuing them to the VSFS cluster. Client consists of three components: 1) Interface; 2) VSFS RPC Module, and 3) Storage Driver. VSFS RPC module handles the communication between client and VSFS back-end servers. Storage Driver enables the underlying storage system to be pluggable to VSFS and handles the storage of the raw file data. Because of the modular design of storage driver, the underlying storage is able to support any existing file system, e.g., HDFS [14], Lustre [38], etc..

Currently VSFS provides the user two interfaces, the POSIX interface and the native VSFS API. The POSIX interface is now implemented through FUSE [3] for fast prototyping purposes. This FUSE module manages the interpreting of the virtual directory into file queries and takes charge of filling the directory with appropriate file-search results. In the future we plan to implement this layer through pNFS [7] for better performance. The VSFS API offers the user the capability to directly utilize VSFS.

Data Flow. In order to better illustrate how VSFS works, we describe the VSFS data flow for file indexing and search operations here, as shown in Figure 3:

File Indexing. When a HPC application wants to index files in VSFS, the VSFS client will use the *(index name, file path)* pairs to ask the master server to locate the index servers to which these files belong (Step 1). Master server will search through its file to index mapping to find which indices these files belongs to, then it searches through the consistent hashing ring and finds out which index servers the indices are located in. After the master server returns the index servers locations to the client (Step 2), the client then parallelly issues requests to the corresponding index servers to perform indexing operation on the specific indices (Step 3).

File Search. When an HPC application wants to search files through VSFS, the VSFS client will parse the search query into VSFS operations. Similar to the indexing workflow, the client needs to ask the master server to identify the index servers to which the indices belong (Steps 1 and 2) and then parallelly queries the corresponding index servers (Step 3). The index servers will return the ID of the files (Step 4), which the client will use to find which metadata servers it belongs to and to contact the corresponding metadata servers (Step 5). The metadata servers will finally return the file paths to the client (Step 6).

Data Placement. In order to make it load balanced and scalable, VSFS utilizes Consistent Hashing (C.H) [43,44] for data placement. C.H is known to have minimum overhead when dynamically adding nodes to or removing nodes from the consistent hash ring and can provide incremental scalability to the system [29,55]. In VSFS, consistent hashing is used for the following two different purposes:

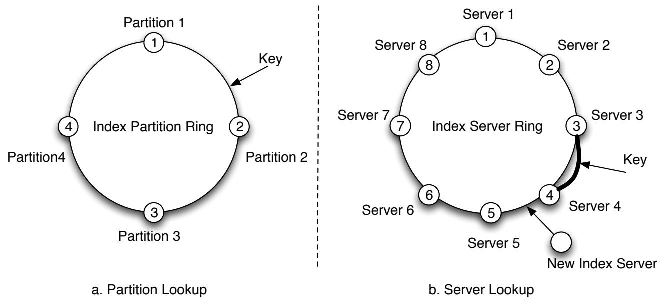

Figure 4: Consistent Hashing Rings used by Partition Lookup and Server Lookup.

Partition Lookup. In order to scale the file index incrementally, as well as utilizing parallelism in the cluster, VSFS relies on consistent hashing to separate a large file index into small partitions and distribute them to multiple index servers. Each file index uses a per-index consistent hashing ring to organize its partitions. In order to give priority to the file-indexing operations, the value on this per-index consistent hashing ring is calculated by hashing the file path of each index record, so that the clients are able to locate index partitions by directly computing the requests. For example, as shown in Figure 4.a, when there is a request arriving, the hash value of the key falls into the hash range between Partition 1 and Partition 2, in VSFS, Partition 1 will be responsible for this key, so this key belongs to the first partition of this index. When the number of records in a partition exceeds a certain threshold, the index server will transparently split the partition into two partitions and trigger the *Online Data Migration* process to migrate the split index to another index server.

Server Lookup. The partitioned index is distributed on index servers through an index server Consistent Hash ring, which is managed by the master server. The client will calculate the hash value of the index partition path, which consists of (***root, name, hash separator***), where the *hash separator* is the beginning value of the hash range this partition manages, and find the corresponding index server. In order to balance the load in the cluster, the index servers in the VSFS cluster will evenly divide the entire consistent hash ring, and each index server will be in charge of all the requests issued to the partition that are in its hash range. As shown in an example in Figure 4.b, eight index servers evenly divided the ring into eight hash ranges, and each index server is in charge of one range. When there is a new index server added to the ring, it will evenly divide the hash range between Servers 4 and 5. When there is a request arriving, the master server will check which range the request key falls into and respond with the corresponding index server information.

The distributed architecture makes VSFS an incrementally scalable system without the capacity limitation of a single machine.

3.3 Implementation Details

Besides the scalable distributed architecture, there are several other important features that make VSFS a fast but durable system. Here we discuss the implementation details of these features:

Durability & Recovery. In order to address the volatile nature of DRAM-based storage, while maintaining low capacity overhead, we use write-ahead log [37, 57] to ensure the durability of VSFS. Each operation on the index will be executed if and only if it has been written into a write-ahead log in memory, which will be flushed to the local disk every 5 seconds. If there is a node failure, then VSFS can replay the operations in the write-ahead-log to reconstruct the previous state of the node. In our current prototype implementation of VSFS, the write-ahead log is located on a shared storage (Lustre). As part of our future work, we also plan to replicate this log to another two replica nodes so that even if the current node fails, the state can still be reconstructed from the other two replicated logs [19, 55].

Online Data Migration. As an index grows to overwhelm the RAM capacity, it is necessary to split the index into two smaller ones and migrate the split index to another machine that has sufficient available storage space. We designed an online data migration mechanism that supports data migration on the fly without impacting the availability of the system. The live migration mechanism is similar to the one used in FAWN [12], which includes two phases: *Data Pre-Copy* and *Merge and Join.* In the pre-copy phase, while the migrated data is being transfered, the old node still handles the requests, but also forwards the update requests to the buffer of the new node. After the data transfer is finished, the new node will merge the buffered updates with the migrated data, and notify the master server to join the server ring. This mechanism ensures that concurrent indexing/search requests are handled consistently during the data migration process.

Caching Consistent Hashing Ring. To offer transparent file-indexing operations, VSFS does not require a user to specify the *root* path of the file index when the users inserts file-index records. Instead, when a user specifies the path of a file being indexed and the index name, the master server recursively looks up its parent directories to find the matched root path for the targeted index and to return to the corresponding index servers. This operation is relatively expensive for the master server. However, because file-level workloads have strong locality [11, 46] and a file-index manages a large sub-namespace under the *index root* directory, it is able to leverage this locality to reduce the load on the master server by caching the information of the file-index root paths as well as the C.H ring of this index on the client side.

4. EVALUATIONS

We evaluate the performance of the VSFS prototype using representative datasets and workloads. In the experiments, we examine the performance in terms of *file-indexing* performance, *file-search* performance, *query scalability*, *I/O performance* and *application performance*, in order to assess how effectively VSFS will likely perform in a real environment.

Experimental Setup. We prototype VSFS on a 20-node heterogeneous Linux cluster testbed to measure its scalability, where 1 node runs as Master Server node, 16 nodes run as Index Severs, and the remaining 3 nodes run as Metadata Servers. 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 4 $\sim$ 8GB RAM. Each node is equipped with a Western Digital WD800JD-75MSA3 80GB 7,200 RPM HDD formatted as Ext4. 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, as shown in Figure 5. As a result, we are able to use the production HPC cluster as clients to stress VSFS when it is necessary. Each node runs Scientific Linux 6.3 and uses the primary disk to store the experimental data. We compare VSFS against the MySQL Cluster Linux (x86, 64-bit) version 7.2.10, which is one of the most widely used open-source SQL databases, and HBase 0.94.6 [21], which runs on top of Hadoop 1.0.4 [13], the *de facto* enterprise Big Data analytics platform. Finally, we also evaluate the I/O overhead that VSFS adds to a production Lustre file system.

To make the performance comparison as fair as possible, we optimize the MySQL and HBase clusters to the best of our knowledge. The MySQL cluster is enabled with Auto-

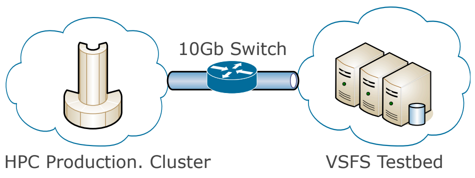

Figure 5: VSFS Testbed

Sharding for write-scalability. Its management node and SQL node run on the same node as VSFS's Master Server does, and its data nodes run on the same nodes as VSFS's Index Server nodes. The HBase cluster runs in a similar configuration, that is, its Master node runs on the same machine as VSFS' master server does, and RegionServers run on the same machines as VSFS's index servers do. To achieve the best write performance, we only use 1-replica of data in both the MySQL cluster and the HBase/Hadoop cluster. Moreover, each node in the MySQL and HBase clusters is configured to use 80% of its physical RAM as MySQL/HBase's buffer to maximize their memory utilization. We configure two different SQL schemas for the MySQL cluster, one with a single SQL table to store all file index records (denoted mysql(s)) and the other with a separate SQL table for each file index (i.e., partitioned table, denoted mysql(p)). For HBase, since each table in it only has one *rowkey* that can be used to retrieve and scan values, we create one separate table for each file index in HBase and use the *file key* in the file-index records as the *rowkey* in each HBase table.

4.1 Index Performance

We first compare the file-indexing performance of VSFS against the MySQL cluster and the HBase cluster on the same testbed. To stress the targeted systems (e.g., VSFS, MySQL and HBase), we use the SLURM scheduler [45] to acquire 30 physical nodes from the production HPC cluster, where each node runs 4 client processes and each client process sends file-index records to two individual file indices. Therefore, there are a total of 120 clients sending requests to a total of 240 indices. Additionally, we use MPI barrier [31] to synchronize the start time of issuing file-indexing requests for all clients. In each test, the clients issue a total of 10 million records, which represents a typical scale of the input files for the MVD analytics application [60].

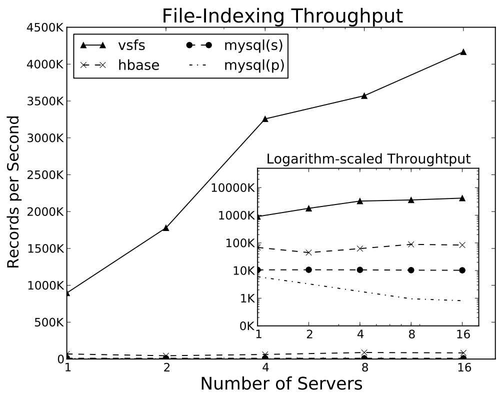

Figure 6: File-Indexing Throughput. In this figure, "*mysql(s)*" represents the throughput of MySQL with a single SQL table and "*mysql(p)*" represents the throughput of MySQL with partitioned SQL tables.

As illustrated in Figure 6, VSFS scales significantly better than both the HBase cluster and the MySQL cluster. VSFS outperforms HBase on all cluster sizes, from a factor of 12× on a 1-node cluster to a factor of 49× faster on a 16-node cluster. It outperforms MySQL even more significantly, from 85× on a 1-node cluster to 408× faster on a 16-node cluster when a single SQL table is used and from 1492× on a 1-node cluster to 4709× on a 16-node cluster when partitioned SQL tables are used. The reason behind the degrading throughput in the partitioned MySQL cluster is that the MySQL cluster 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. However, because of the sharding-capability being enabled for the MySQL cluster, each SQL table is partitioned and distributed to all data nodes. The SQL node needs to pull data from multiple data nodes to perform this index-table-locating task. Therefore, with the same amount of data being pulled from data nodes, the more the data nodes, the lower the throughput the MySQL cluster can achieve. When a single table is used to store indices in the MySQL cluster, the bottleneck is apparently shifted to the CPU on the SQL node, where the CPU utilization is observed at $60-80\%$ during all the evaluations. The reason behind this high CPU utilization is that all SQL queries must go through the single SQL node before sending the write requests to the data nodes, because MySQL does not support distributed locking on a single 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. Because the meta-table is very small in size, usually only hundreds of KBs, it is highly likely that the table scanning operations take place in a single RegionServer. As a result, the throughput is limited by the CPU of this RegionServer. Finally, in VSFS, when a client issues an update request, the client first queries the master server to locate all index servers containing the partitions of the corresponding index, and caches the index server location information locally on the client's machine during the execution. As a result, all the subsequent update requests to the same index only need to be calculated on the client side to determine which index servers to communicate with. Because there exists strong space locality in the file system workloads [11], especially the file-indexing workload that is usually performed by a single HPC/Big Data analytics application, by exposing this parallelism, VSFS is able to gain 12× $\sim$ 4709× speed up over the two baseline production solutions: HBase and MySQL.

4.2 Search Performance

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

The first evaluation is conducted to compare the search performance among VSFS and two other file-search approaches of MySQL and HBase. In all tests, a single client issues one file-search request of *"gathering* $10,000$ *files from the same index"*, which is a common request for the MVD analytics. The end-to-end latency of the file-query request is measured, as shown in Figure 8. VSFS is up to 6275× faster than HBase and 102× faster than MySQL with partitioned

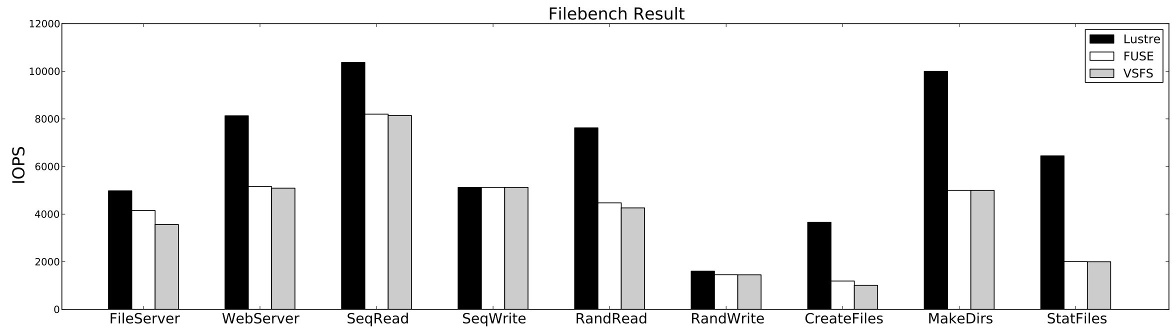

Figure 7: Filebench Results. VSFS, as a FUSE-based file system, only introduces small additional overhead to the inherent FUSE overhead. The additional VSFS overhead comes mostly from the file creation and deletion operations.

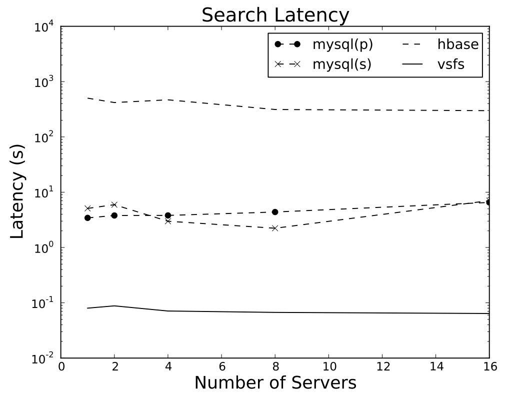

Figure 8: Search Latency As A Function of the Number of Index Servers

SQL tables and 110× faster than MySQL with a single SQL table. This significant performance advantage stems from our RAM-based design that enables in-memory processing for VSFS. Note that both HBase and MySQL use the same amount of RAM space as VSFS in all tests.

Number of Servers	1	2	4	8	16
95th Percentile	0.064s	0.053s	0.052s	0.046s	0.046s
99th Percentile	0.507s	0.073s	0.066s	0.069s	0.068s
Average	0.036s	0.019s	0.019s	0.018s	0.019s
Peak	0.674s	0.432s	0.525s	0.385s	0.340s

Table 3: Distribution of Stress-Tested File-Search Latency on VSFS

We conduct the second evaluation to stress VSFS by issuing simultaneous file-search requests in an open-loop manner. In this test, we use SLURM to 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, of which each asks for 1000 files from 20 file indices in an open loop. The distribution of the latency for every request is measured, as listed in Table 3. Although we were not able to acquire more clients-hosting physical nodes to stress VSFS due to the limited resource availability of the heavily commissioned production HPC cluster, this evaluation still demonstrates VSFS' capability of serving bursty requests from hundreds of clients simultaneously. As a result, this low-latency file-search operation makes it feasible for VSFS to be deployed directly as a file-system service in data-intensive environments, such as HPC and Big Data environments.

4.3 I/O Overhead

The current VSFS prototype implements a POSIX layer through FUSE to demonstrate its versatile searchable namespace. In order to separate VSFS's overhead from that of FUSE, we implemented a *FUSE-based file system* that only passes requests to the underlying file system. We use a production Lustre parallel file system as our baseline and mount the FUSE-based file system and VSFS on top of Lustre. 9 pre-defined Filebench [64] workloads, including 2 macro workloads (fileserver, webserver) and 7 micro workloads, are used to measure the overall overhead of VSFS. The total number of files of each workload is 100000 and each file has a mean file size of 4KB, the I/O size of the test is 1KB. Each workload is configured to run with 16 threads and the test runs for 60 seconds.

As shown in the Figure 7, for macro workloads, VSFS introduces an additional 12% degradation for *fileserver* and an additional 0.9% degradation for *webserver* to the pure-FUSE implementation. With the micro workloads, VSFS contributes 0.7% and 4% additional overheads in the sequential read and random read workloads, respectively. However, for the sequential and random write workloads, there is nearly no additional overhead introduced by VSFS. For the metadata-intensive workload *CreateFiles*, only 17% additional overhead is introduced because file creation and deletion are the only operations by which a VSFS client needs to contact the VSFS cluster in this benchmark. Finally, there is no significant additional overhead introduced by VSFS for the *MakeDirs* and *StatFiles* benchmarks.

In summary, the FUSE-based prototype of VSFS is shown to incur relatively low I/O overhead, suggesting that VSFS can be a feasible and potentially powerful enabling tool for data-intensive analytics applications in HPC and Big Data environments. We also plan to implement a pNFS [7] client for production usage, which is expected to achieve a native performance comparable to the current Lustre and other NFS clients.

4.4 Application Performance

We use Molegro Virtual Docker (MVD) [60] as the example to demonstrate how VSFS may benefit real HPC/Big Data analytics applications [5,24,42]. Due to our lack of privileges to mount a FUSE-based file system on the production HPC

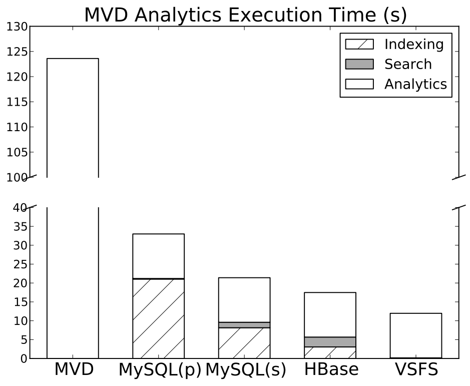

Figure 9: MVD Application Performance

cluster, all tests in this evaluation run on our 20-node Linux cluster testbed. We configure a 5-node subcluster to run the file-search services (i.e., MySQL/HBase/VSFS), and use the remaining 15 nodes to run the MVD program. Each run of the MVD program reads a $\sim$ 4KB file as input, computes the data and then writes the output file ($\sim$ 4KB) back to the Lustre file system in the production HPC cluster. Each run takes approximately 7 seconds in the production HPC cluster. As a result, to emulate the high workload intensity of running the MVD program on a 1000-node HPC cluster that would presumably be $1000/15 = 67$ times faster than the 15-node testbed cluster, we proportionally reduce the computation time (i.e., the think time) of the MVD program by a factor of $1000/15 = 67$.

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 and have already been imported to the system. Before running the experiments, $10,000$ file-index records are generated. In the original MVD case (i.e., denoted by "*MVD*" in Figure 9), 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 four cases, the MVD analytics application will utilize the external file-search services provided by MySQL (both the "s" and "p" versions), HBase and VSFS respectively to filter out unrelated data. Therefore, in each of these four cases, the file-search service first indexes the $10,000$ file records obtained from previous runs and then searches for and filters out $1,000$ 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 Figure 9, with the help of the external file-search service, the execution time of the MVD analytics can be significantly reduced. However, Table 4 clearly shows that the two MySQL-based solutions and the HBase-based solution add significant indexing and search latencies to the total processing time of the MVD analytics program, while VSFS' s indexing and search latencies are very insignificant compared to the MVD computation (analytics) time. In this evaluation, running MVD on VSFS is up to 2.6× faster than running on the MySQL-based solutions, 1.5× faster than the HBase-based solution and 10.3× faster than the original

Solution	Index	Search	Analytics	Total	SLOC
MVD	N/A	N/A	123.6s	123.6s	0
MySQL(p)	18.842s	0.162s	11.8s	30.804s	405
MySQL(s)	8.151s	1.45s	11.8s	21.401s	411
HBase	3.63s	2.615s	11.8s	18.035s	391
VSFS	0.127s	0.043s	11.8s	11.97s	0*

Table 4: A Breakdown of the MVD Processing Time into Indexing, Search and Computation (Analytics) and 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, as shown in Example 2 on page 5.

MVD solution. It is worth mentioning that in the MySQL case with a single table (i.e., "*MySQL(s)*"), the latencies of inserting and searching files will dramatically increase on larger data-sets (i.e., 10-billion MVD results) because all file-index records are stored in a single SQL table. Additionally, the extra Source Line of Code (SLOC) required by each solution to integrate its file-search service into the original MVD analytics program in our tests is also listed in Table 4. While the extra SLOC results for test cases of the MySQL and HBase solutions listed in Table 4 are most certainly much smaller than their likely values in a production system, they still implicitly illustrate the significant amount of modifications and the background knowledge (e.g., SQL syntax, SQL optimization, key-value data models, etc.) required to efficiently utilize those systems. In contrast, using VSFS does not require any additional knowledge other than basic file system operations of the POSIX API.

In summary, VSFS is capable of transparently and significantly accelerating the HPC analytics applications (e.g., MVD) without requiring domain experts to master storage/-database knowledge and modify the legacy code.

5. CONCLUSION AND FUTURE WORK

This paper presents VSFS, a novel storage model to address the need for a transparent file system level search capability for HPC/Big Data analytics, and its prototype, a searchable distributed file system with real-time file-search/file-indexing capability. By offering a POSIX compatible searchable namespace and a scalable DRAM-based architecture, VSFS offers the capability to transparently and significantly accelerate the processing of the HPC/Big Data analytics applications. The prototype evaluation shows that VSFS outperforms MySQL-based and HBase-based file-search solutions by 2$\sim$3 orders of magnitude in the file-indexing and file-search performance with acceptable I/O overhead. It is also demonstrated to be capable of dramatically accelerating a real world HPC analytics application (MVD).

After demonstrating the VSFS prototype to domain experts, they have shown strong interests in integrating VSFS into their analytics applications. Furthermore, VSFS will be open sourced and publicly accessible in the near future. We also plan to optimize the performance of VSFS by implementing pNFS to provide the POSIX searchable namespace, to use a master server cluster. More advanced analytics functionalities in the searchable namespace are planned for design and implementation as well.

6. REFERENCES

[1] Castor: Cern advanced storage manager. `http://castor.web.cern.ch`.

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

[3] Filesystem in userspace. `http://fuse.sourceforge.net/`.

[4] Greenplum. `http://www.greenplum.com`.

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

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

[7] Parallel NFS. `http://www.pnfs.com/`.

[8] Root | a data analysis framework. `http://root.cern.ch`.

[9] Teradata aster. `http://www.asterdata.com`.

[10] VoltDB. `http://voltdb.com`.

[11] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. In *FAST '07*, 2007.

[12] 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.

[13] Apache.org. Apache hadoop. `http://hadoop.apache.org/`.

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

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

[16] B. Azvine, Z. Cui, D. Nauck, and B. Majeed. Real time business intelligence for the adaptive enterprise. In *E-Commerce Technology, 2006. The 8th IEEE International Conference on and Enterprise Computing, E-Commerce, and E-Services, The 3rd IEEE International Conference on*, pages 29–29, 2006.

[17] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

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

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

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

[21] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 1071–1080, New York, NY, USA, 2011. ACM.

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

[23] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. OSDI '06, Berkeley, CA, USA, 2006. USENIX Association.

[24] M. H. Chin, M. J. Mason, W. Xie, S. Volinia, M. Singer, C. Peterson, G. Ambartsumyan, O. Aimiuwu, L. Richter, J. Zhang, I. Khvorostov, V. Ott, M. Grunstein, N. Lavon, N. Benvenisty, C. M. Croce, A. T. Clark, T. Baxter, A. D. Pyle, M. A. Teitell, M. Pelegrini, K. Plath, and W. E. Lowry. Induced pluripotent stem cells and embryonic stem cells are distinguished by gene expression signatures. *Cell Stem Cell*, 5(1):111 – 123, 2009.

[25] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. SoCC '10. ACM, 2010.

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

[27] 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*, pages 213–229, New York, NY, USA, 1965. ACM.

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

[29] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

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

[31] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[32] D. Giannone, L. Reichlin, and D. Small. Nowcasting: The real-time informational content of macroeconomic data. *Journal of Monetary Economics*, 55(4):665 – 676, 2008.

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

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

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

[36] B. Gopal and U. Manber. Integrating content-based access mechanisms with hierarchical file systems. In *OSDI '99*, 1999.

[37] R. Hagmann. Reimplementing the cedar file system using logging and group commit. *SIGOPS Oper. Syst. Rev.*, 21(5):155–162, Nov. 1987.

[38] 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. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12,

pages 6:1–6:8, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[39] 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.

[40] 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.

[41] 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.

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

[43] 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. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[44] 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, New York, NY, USA, 1999. Elsevier North-Holland, Inc.

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

[46] 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*, 2009.

[47] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin. The sequence alignment/map format and samtools. *Bioinformatics*, 25(16):2078–2079, Aug. 2009.

[48] L. H. Marcial and B. M. Hemminger. Scientific data repositories on the web: An initial survey. *Journal of the American Society for Information Science and Technology*, 61(10):2029–2048, 2010.

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

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

[51] N. Marz. Storm project : Distributed and fault-tolerant realtime computation. `http://storm-project.net/`.

[52] 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.

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

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

[55] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

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

[57] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, 2005.

[58] C. A. N. Soules and G. R. Ganger. Connections: using context to enhance file search. *SIGOPS Oper. Syst. Rev.*, 39(5), 2005.

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

[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] Y. Wang, C. Barbacioru, F. Hyland, W. Xiao, K. Hunkapiller, J. Blake, F. Chan, C. Gonzalez, L. Zhang, and R. Samaha. Large scale real-time pcr validation on gene expression measurements from two commercial long-oligonucleotide microarrays. *BMC Genomics*, 7(1):59, 2006.

[62] 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.

[63] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the panasas parallel file system. In *FAST '08*, 2008.

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

[65] 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.