2014

# Propeller: A Scalable Real-Time File-Search Service in Distributed Systems

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

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

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

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

2014 IEEE 34th International Conference on Distributed Computing Systems
Pages: 378 - 388, DOI: 10.1109/ICDCS.2014.46

# Propeller: A Scalable Real-Time File-Search Service in Distributed Systems

Lei Xu Hong Jiang Lei Tian Ziling Huang
University of Nebraska Lincoln
{lxu,jiang,tian,zhuang}@cse.unl.edu

***Abstract*—File-search service is a valuable facility to accelerate many analytics applications, because it can drastically reduce the scale of the input data. The main challenge facing the design of large-scale and accurate file-search services is how to support real-time indexing in an efficient and scalable way. To address this challenge, we propose a distributed file-search service, called *Propeller*, which utilizes a special file-access pattern, called *access-causality*, to partition file-indices in order to expose substantial access locality and parallelism to accelerate the file-indexing process. The extensive evaluations of *Propeller* show that it is real-time in file-indexing operations, accurate in file-search results, and scalable in large datasets. It achieves significantly better file-indexing and file-search performance (up to $250\times$) than a centralized solution (MySQL) and much higher accuracy and substantially lower query latency (up to $22\times$) than a state-of-the-art desktop search engine (Spotlight).**

## I. Introduction

Many analytics applications [9], [16], [32] run on top of file systems, since file systems provide performance features that are by and large unmatched by database solutions [9], [14], [27], [31], [39], [43]. However, compared to databases, file systems fall short of providing flexible data retrieval capabilities: *the static file path scheme is incapable of adapting to various data retrieval demands* [19], [33].

File-search service, which helps applications retrieve desired files out from larger dataset, should be an ideal solution to accelerating such analytics applications by reducing the scale of input data [33], [44] (i.e., data filtering). Unfortunately, the existing file-search services [11], [20], [25], [26], [30] are neither scalable for nor capable of being deployed in data-intensive environments. For example, the crawling-based file-search engines[11], [30] introduce inevitable and non-negligible crawling delays in updating index, which leads to unpredictable accuracy of file-search results.

Serving file-search requests for analytics applications, especially the time-critical ones, in large-scale systems impose several unique challenges that have not been well addressed by previous studies and existing solutions[19], [22], [25], [26], [30], [41]:

- The file-search results must be strongly consistent with the file content. This is because, unlike the web search engines [16], [40] or the desktop search engines [11], [20], [30] where human users can usually tolerate inaccurate or outdated results to some extend, many analytics applications cannot tolerate such inaccuracy or staleness [4], [8], [31],
- The file-indexing overhead must be small, because the file indices must be frequently updated to be consistent with the file content. However, intensively updating file indices is costly and usually impractical for data-intensive systems [26], [30].

Clearly, the most critical requirement for such a real-time file-search service is *to keep file indices always up-to-date* (a.k.a., the *inline* file-index model) in a large-scale data-intensive system. The high overhead of keeping strong consistency between the file indices and file contents stems from the increasing scale of file index (i.e., the number of files) and the file re-indexing triggered by the continuous file updates. While techniques have been proposed to reduce the file index scale [25], [30], they fail to keep file index always up-to-date or overcome the performance bottleneck resulting from continuous file index updates.

To this end, we propose a distributed file-search service, called *Propeller*, to offer real-time file-indexing and file-search functionality in data-intensive environments. Propeller is specially designed to speedup file-indexing operations to ensure the freshness and timeliness of file-search results so that they fully reflect the latest changes to files. Therefore, the file-indexing operations are on the I/O critical path to ensure the freshness of index content, while the file search requests that are each capable of accomplishing the work of a huge number of "readdir" operations are relatively rare in real-world workloads [6]. For example, log analytic workloads [31] can index petabytes of logs in real-time before dozens of ad-hoc queries issued by either data scientists or applications. Propeller's real-time indexing scheme is designed based on the observation that file accesses of analytics applications tend to frequently cluster amongst and around correlated files. To effectively leverage the access locality exposed from this application-aware file-access behavior, Propeller introduces *Access-Causality Graph* (ACG), which represents the files (i.e., vertices) access causal relationships (i.e., edges), to capture and exploit the file-access patterns. ACG enables Propeller to *automatically partition* the large file indices into smaller ones while preserving access locality by applying graph partitioning algorithms [24], [28], [37] to confine the index updates to a few smaller indices (i.e., sub-graphs).

This paper aims to make the following contributions:

1) The development of a real-time distributed file-search service prototype, Propeller, with the real-time file-

1063-6927/14 $31.00 © 2014 IEEE
DOI 10.1109/ICDCS.2014.46

indexing capability, enabled by its novel index partitioning technique, *Access-Causality Graph* (ACG), which is designed to effectively address both the index scalability and the intensive index update challenges.

2) The extensive evaluation demonstrating Propeller's feasibility and efficacy in data-intensive environments. The Propeller prototype significantly outperforms a centralized SQL database solution (MySQL) in file-indexing and file-search, and offers better file-search latency and accuracy than a state-of-the-art desktop search engine (Spotlight), especially under write-intensive I/O workloads.

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

## II. Related Work, Background and Motivation

Given the explosively growing volume of data stored in the file systems [9], [14], [23], [27], [31], [39], efficient and flexible file-search solutions have been recognized as an essential service for end-users and system administrators alike [11], [20], [25], [26], [30], [35]. Furthermore, many analytics applications can greatly benefit from utilizing these file-search services to accelerate their computations by filtering out most of the input data. For instance, Molegro Virtual Docker (MVD) [45], a computational drug-discovery application, stores the full structure information of a particular protein in a single input file. Its protein-structure dataset typically is very large ($10^7 \sim 10^8$ files), and there are hundreds of different attributes from each protein (i.e., structures or energy characteristics). With a file-search service, the MVD application can continuously compute a smaller and refined set of proteins that share similar characteristics observed from the previous computation to evaluate the effectiveness of a new drug. Unfortunately, existing file-search solutions are not designed nor adequate for serving such analytics applications in large-scale data-intensive environments.

Analytics applications [16], [32], instead of human end-users, require a file-search service to return real-time results that are always accurate and up-to-date (i.e., consistent with all the file contents within the file system), so that the analytics applications can immediately process these data with confidence [31]. Therefore, it requires files being re-indexed immediately (i.e. real-time) after their contents have changed. Nonetheless, since the current practice of file-indexing is crawling based and the file indexing is done in the background (i.e., *offline indexing*), the indexing overhead can be hidden from the I/O critical path [11], [20], [30]. This practice, however, cannot guarantee the accuracy or the freshness of the file-search results for an obvious reason: the inevitable and often significant delay from when a change is made to a file to when the file's index is updated, caused by the asynchronous crawling process, makes the file indices always outdated.

In order to demonstrate the inaccuracy introduced by the asynchronous crawling process, we use Spotlight [11] as the test platform to evaluate how the continuous updates impact the accuracy, or *recall* [5], of file-search results. Although only running on a single machine, Spotlight shares the crawling essence of the other distributed solutions [21], [30], to which we do not have access. Therefore, Spotlight is arguably adequate and convincing in exposing the inaccuracy of file-search results for a class of file-search services [11], [21], [30], [35].

In this demonstration, the recall of file-search results is measured as a function of the background I/O intensity, denoted by the number of files copied per second (FPS). The measure of recall is defined to be the fraction of relevant files that are returned as file-search results. As illustrated in Figure 1, the recalls of Spotlight are low ($< 53\%$), because it only supports limited pre-defined file types and thus it cannot include all files in the test dataset, which consists of 10 workstation and virtual machine file system images, and sensitive to the intensity of background file copying. With highly intensive file copying (e.g., $> 10$ FPS in this test), the re-indexing process in Spotlight is so frequently triggered to update the index that the recall values are dropped to 0 during re-indexing! It is worth mentioning that the desktop search engines like Spotlight and Google Desktop Search integrate the file-system notification mechanisms [10], [34], which enable them to respond much faster to the new file modifications than the distributed search appliances do. Additionally, the I/O intensity in large-scale data-intensive environments will be orders of magnitude higher than what has been shown in this test. As a result, it is reasonable to expect that the inaccuracy of the file-search results is inevitable for the asynchronous crawling-based solutions [21], [30].

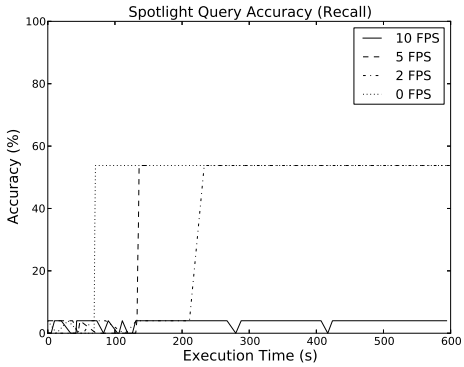



Fig. 1. **Recall Values of the Spotlight Search Results**. *FPS*: file-copy operations per second. After completing the Spotlight index rebuilding, we immediately spawn a background process to copy files at various speeds and a foreground process to continuously send queries to Spotlight. 0 FPS means that there is no background process.

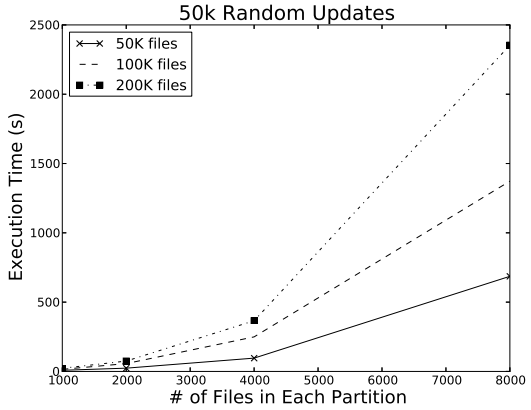
Evidently, the real-time file-indexing capability is a prerequisite to guaranteeing the accuracy and freshness of the file-

search results, which further enables the analytics applications to utilize the file-search service to accelerate computing. However, applying *real-time file-indexing* on large-scale systems is difficult, because maintaining large-scale index usually results in poor indexing performance and adds considerable overhead along the I/O critical path. Therefore, to address these challenges, we propose Propeller, a highly scalable distributed file-search service that is designed to provide real-time file-indexing performance in data-intensive environments.

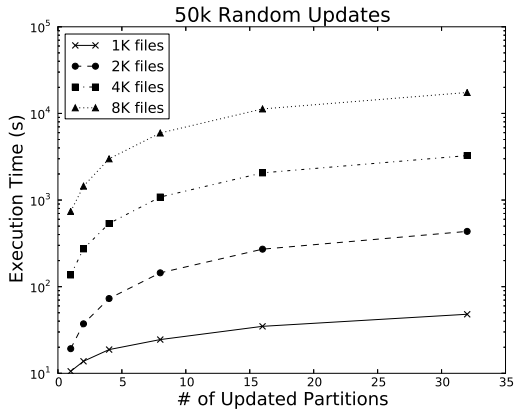
## III. Access-Causality Based File Partitioning

The primary obstacle to real-time file indexing in large-scale file systems is the poor scalability of the costly index-updating operations. A common remedy for this scalability problem has been to partition the index to narrow down the scope of operations. Existing solutions are either namespace-based [30], [38] or file-metadata attribute-based [25] partitioning, which are all based on *static file attributes* (e.g., file location or file metadata). However, our analysis and ongoing experiments suggest that *partitioning based on the static file attributes can result in significant traffic to the I/O critical path due to the frequent real-time file-indexing operations*.

To better understand the performance impact of partitioning schemes, we develop a program to conduct a sensitivity study of partition scale and inter-partition updates on one machine. It simulates a typical application issuing 50,000 writes to partitions of files to trigger inline indexing and measures the execution time as a function of partition size and of access concentration (inter-partition accesses). Each partition maintains three file indices on HDDs: a B+tree, a Hash Table and a K-D-Tree [12]. As shown in Figure 2(a), 50,000 file update requests are randomly distributed to a fixed total number of files that are evenly partitioned into groups of a given size, which ranges from $1,000$ files per partition to $8,000$ files per partition. For each configuration, the experiment runs 3 times and the average result is measured. The evaluation results clearly demonstrate that a larger group size leads to worse indexing performances. In the second test, 50,000 updates are issued to an increasing number (i.e., 1∼32) of partitions in a given partitioning scheme (i.e., a given group size), to evaluate the impact of the inter-partition accesses, or access concentration. The result, shown in Figure 2(b), indicates that the number of accessed partitions significantly impacts the indexing performance as well. More specifically, the higher the access concentration is, the higher the inline indexing performance will be. The key takeaway from these experimental observations is that *not only the partition scale, but also multi/cross-partition accesses, have a significant performance impact on file-indexing operations*. Unfortunately, this *cross-partition accesses* cannot be observed and controlled from the static file attributes (e.g., file location or file metadata). For instance, we observed that programs usually access files located at various physically separated directories, which are highly likely to be located in different namespace-based partitions [30], [38], [47]. Figure 3 shows that a Linux Firefox web browser accesses the "bin" directory, "log" directory, "home"




(a) **Impact of Partition Size**. Randomly accessing the same number of files that are partitioned into different number of equally-sized groups. Larger partition leads to lower update performance.




(b) **Impact of Inter-Partition Access (log-scale)**. Randomly accessing different number of groups ($1 \sim 32$) with the same group size. More inter-partition updates (i.e., updates involving a large number of partitions) lead to lower update performance.

Fig. 2. Performance Impacts of Partition Size and Inter-Partition Accesses.

directory, etc. during its execution. Additionally, many big data datasets have large fan-out directories, in which there is an enormous number of files in the same directory [9], [31], [32], [46]. Both of the aforementioned examples make it difficult, if not impossible, for the existing partitioning approaches to reduce the prohibitively costly inter-partition updates.

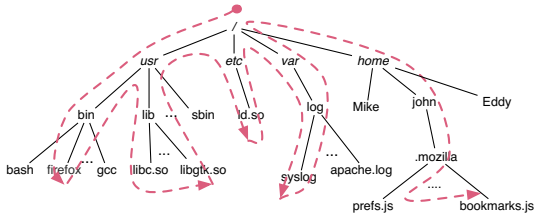



Fig. 3. Firefox Dataflow

Consequently, to efficiently perform real-time file-indexing operations, the partitioning scheme of file index must limit

the scale of the partition while reducing inter-partition IOs. The file-access patterns, considered the *dynamic file attributes*, must be taken as a significant partitioning criteria. Unsurprisingly, we have found that it is applications that determine their accessed file sets and corresponding file-access patterns. This suggests that *file sets may be naturally partitioned by virtue of semantic and access correlations of applications*. For instance, Table I, which summarizes the file-access patterns we monitored from the executions of four commonly-used applications on a Linux machine, clearly indicates that any two different applications share very few files, implying that file accesses are highly application-oriented and application-isolated. This observation can also be extended to many classes of analytics applications [13], [32].

| Program Execution | Apt-get | Firefox | OpenOffice | Linux Kernel |
|---|---|---|---|---|
| Accessed Files | 279 | 2279 | 2696 | 19715 |
| Apt-get | N/A | 31 (1.36%) | 62 (2.29%) | 29 (0.15%) |
| Firefox | 31 (11.1%) | N/A | 464 (17.2%) | 48 (0.24%) |
| OpenOffice | 62 (22.2%) | 464 (20.3%) | N/A | 45 (0.22%) |
| Linux Kernel | 29 (10.3%) | 48 (2.11%) | 45 (1.69%) | N/A |

TABLE I
**COMMON FILES ACCESSED BY EXECUTIONS OF DIFFERENT PROGRAMS**: APT-GET[17] (SYSTEM MANAGEMENT), FIREFOX (WEB BROWSING), OPENOFFICE (DOCUMENT EDITING) AND LINUX KERNEL BUILDING.

To this end, we propose a distributed file-search service, Propeller, which first captures the file-access correlation, called *access-causality*, and then uses this correlation to partition the files by a partitioning algorithm. Access-causality is defined as the access correlation that represents the causality of the file content. To be more specific, two files $f_A$ and $f_B$ are considered access-causal, denoted by $f_A \rightarrow f_B$, if file $f_A$ is opened by a process $P$ that either reads or writes at time $t_0$ and file $f_B$ is opened by the same process $P$ that writes at time $t_1$, where $t_0 < t_1$. That is, file $f_A$ is considered as the content producer of file $f_B$. Propeller constructs directed *access-causality graphs* (ACGs) from these captured file causalities. In each such graph, a vertex represents a unique file and a weighted edge connecting two vertices represents the access causality between two files, that is, the number of times these two files are opened by the same process in the defined order. Figure 4 illustrates the process of updating the ACG during a program's execution.

These ACGs have the following beneficial properties that enable Propeller to automatically partition and organize file indices to significantly improve the file-indexing performance.

1) The definition of an ACG guarantees that ACG can accurately predict the possibility of files being accessed together, since it actually represents the execution semantics of applications, which are very stable.
2) The ACGs between two different applications, or even within a single application on two different datasets [9], [13], [32], are only loosely connected or completely disconnected, as observed from Table I.
3) The ACGs captured from a single application are still likely to have several disconnected components, as evidenced and elaborated in Section 4.1.
4) For a connected component of a large ACG, since the weight of an edge is defined as the number of times the two files are accessed together, it is amenable to be further partitioned into sub-graphs with a minimal weight of cut that represents the number of inter-partition accesses.

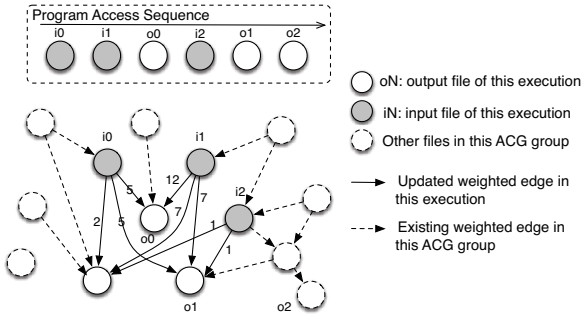



Fig. 4. Updating File Access-Causality Graph

Thus, Propeller is able to partition the files by directly finding the connected components in the ACGs. Note that Propeller clusters small connected components of the ACG from the same application into a single partition to prevent the fragmentation of indices. However, if the scale of a connected component of an ACG grows and exceeds a certain threshold (e.g., $50,000$ files), Propeller is capable of starting a background process to cut the connected component into two sub-graphs that 1) have similar scale and 2) have minimal weight of cut. Therefore, Propeller's partitioning problem can be reduced to the $2 - way$ graph partition problem. Given the existence of several heuristics and approximation algorithms [24], [37], [42] that have been widely used to solve this problem, we choose the METIS [28] algorithm, because it is shown to be very stable and reliable in obtaining approximately equal-sized sub-graphs for our context of the problem.

## IV. DESIGN AND IMPLEMENTATION

In this section we present the design and implementation of a Propeller prototype in a distributed system. As a distributed file-search service, Propeller utilizes ACGs to provide a practical highly-scalable real-time file-indexing facility in data-intensive environments. It is worth noting that Propeller is a general-purpose file-search service, which means that it supports *not only* the indexing of file metadata, such as file size, modification time or user id [25], [30], but also the indexing of arbitrary user-defined attributes on files. Users can define an arbitrary index with a *globally unique name* with the supported index structures (i.e., b-tree, hash table or K-D-tree). Moreover, in order to simplify the development, the Propeller

prototype is organized as a Propeller cluster consisting of one *Master Node*, multiple *Index Nodes*, as illustrated in Figure 5 [9], [18]. Specifically, in order to automatically capture the file access-causality, Propeller's distributed client is implemented under the existing file system on the client side. These distributed components are elaborated in details below.

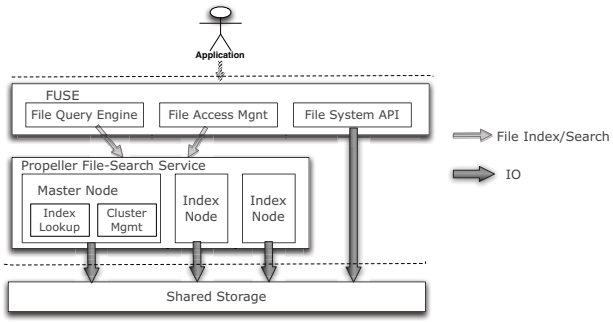



Fig. 5. The Propeller File-Search Service Software Stack

**Client**. In order to transparently capture the file access behaviors, Propeller's client is implemented in a FUSE-based file system [2]. We implement a *File Access Management* module in the client-side FUSE file system to intercept every file open and close operation. The client constructs ACGs from these captured open and close operations in RAM, using the ACG construction algorithm described in Section III. These newly constructed ACGs are initially cached in the client-side RAM and flushed to the Index Nodes after the I/O process finishes. Propeller does not guarantee the consistency for the ACGs to protect against scenarios such as node failures. This is because the inconsistency of an ACG is tolerable since it does not affect the quality (i.e., the accuracy) of file-search results. Choosing a weak consistency model for ACGs also significantly alleviates the I/O overhead of file indexing operations. Additionally, a *File Query Engine* module is implemented as a local RPC service on the client machine, interpreting the file-search requests from either the file system namespace [19], [33] (e.g., a dynamic query-directory "/foo/bar/?size>1m") or a file-search API for applications, and sending the corresponding requests to the Propeller cluster.

**Master Node (MN)** is the central index metadata and coordination server that 1) manages the Propeller cluster, and 2) determines and coordinates how and where the clients send their file-search and file-indexing requests to the corresponding Index Nodes. First, it manages the metadata of indices, such as the locations of ACGs and a hash table that maps from files (i.e., inode) to ACGs identified by the ACG IDs. Second, it maintains the running status of the cluster, such as the location of each ACG, as well as the available resources (e.g., free disk space) on each node. Because it only makes the routing decisions for the file-indexing/search requests, instead of serving the heavy IOs or the actual file-indexing requests, this single Master Server architecture can perform reasonably well in supporting hundreds of Index Nodes [9], [18]. Additionally, the metadata of indices (i.e., file-to-ACG mappings) are periodically flushed to the shared storage to prevent data loss when the server crashes. Finally, as this paper mainly focuses on the index partitioning scheme, designing highly-available Master Node(s) (e.g., to preventi the single point of failure) is beyond the scope of this paper.

**Index Node (IN)** manages the partitioned file indices and services the client's file-indexing or file-search requests. Three categories of index structures are supported at the current stage of the prototype: B-tree, hash table and K-D-Tree. Each ACG can have all three types of file indices, although not all of these indices must be filled with contents. To support user-customized indices, each ACG has a table to point an index name to the actual index within this ACG, and this table is managed by the Index Node. As a result, all file indices within an ACG must be managed by the same Index Node. All the indices, as well as the ACGs and their metadata, are stored as regular files in the underlying shared file system. To reduce the real-time file-indexing latency, Index Nodes aggressively cache the file-indexing requests. When a client sends a file-indexing request, this request is appended to a write-ahead log and inserted into the in-memory index cache. The in-memory cached file-indexing requests are only committed to the index in either of the following events: 1) after a predetermined time interval (also called timeout, e.g., 5 seconds), or 2) upon the arrival of the next file-search request, whichever occurs first. Because file-search requests, as presented as "readdir" operations, are very rare in typical file system workloads [6], the file-indexing cache is shown to be very effective for indexing-intensive workloads. Finally, as shown in Fig 6, each Index Node periodically sends heart-beat requests to the Master Node to acknowledge its runtime status as well as the metadata of ACGs.

**Parallel File-Indexing and File-Search Operations**. As illustrated in Figure 6, a typical file-indexing or file-search request starts from the *File Query Engine* at the client-side, asking MN for the ACGs and their locations (i.e., INs). For update requests (i.e., file-indexing or ACG updates), if the file or the ACG does not exist in MN, MN first allocates the metadata for this new ACG, and then assigns it to the least loaded IN. After the MN successfully locates the ACGs and the corresponding INs, a list of ACGs and INs are sent to clients. Because ACGs are partitioned in a way that significantly reduces the intra-ACG updates, it offers Propeller a great opportunity to send the file-indexing or file-search requests to the selected INs in parallel. Moreover, there is no cross-ACG or cross-IN transaction needed to be maintained. As a result, the clients can process the file-indexing or file-search requests from different applications simultaneously, as illustrated in Figure 6. Finally, for the file-search request, the client-side *File Query Engine* sends the file query requests to all INs, which hold the ACGs that have the indices with the given globally unique name, and each IN issues the query to these ACGs, then the client aggregates the file names returned from these INs.

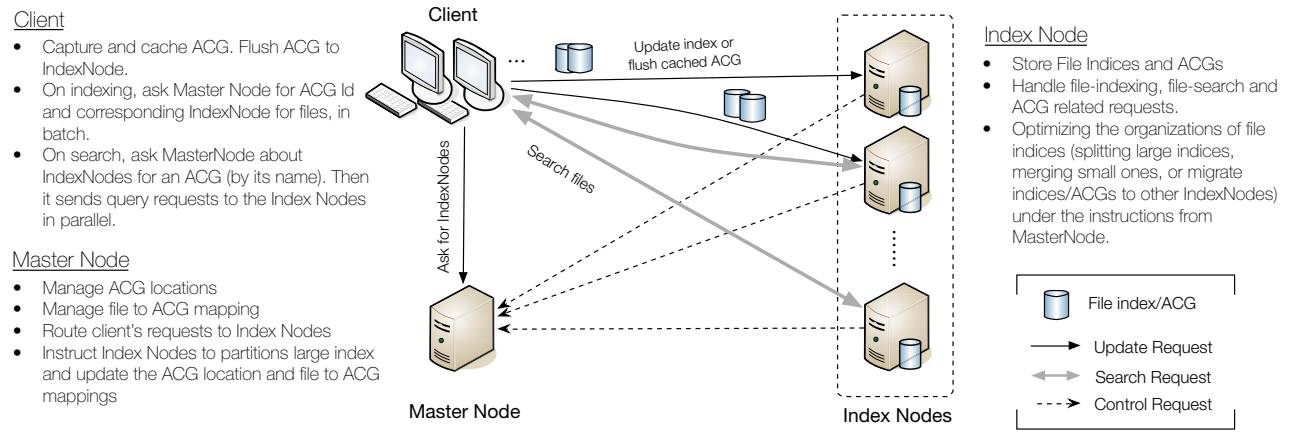



Fig. 6. Propeller's Distributed Architecture

**Workflow**. As mentioned above, Propeller is implemented as a prototype of a general file-search service, which supports indexing and searching the fields that are not limited to the inode metadata (i.e., size, uid or mtime) [25], [30]. Therefore, users or applications need to first create a customized file index with a unique name, for the convenience of future operations. To this end, Propeller is able to *autonomously* manage the location and scale of ACGs for performance optimization, because all ACG performance criteria are observable during the execution of the applications. For instance, the client-side *File Access Management* module captures file-creation and file-deletion operations and updates the file to the ACG mapping in the MN accordingly. For another example, when it observes that the scale of an ACG exceeds a certain threshold, the IN initializes a background ACG-splitting task and sends acknowledgement to MN. MN assigns the newly partitioned ACG a new IN, and instructs the original IN to migrate the split ACG to the new IN. Additionally, the contents of each index are fed directly by users or applications. It is worth noting that the file-indexing and file-search operations are not on the I/O critical path, because users and applications can choose when to update the indices. Eventually, the file raw data and file metadata are managed by the underlying shared storage (Figure 5), with the exception of the mapping from files to ACGs that must be managed by MN. Thus the raw file system metadata and I/O operations do not increase the stress on Propeller either.

## V. Evaluations

We evaluate the performance of the Propeller prototype using representative datasets and workloads. In the experiments, we examine the performance metrics in terms of *file-indexing* performance, *file-search* performance, *query accuracy*, *query scalability*, and *system overhead*, in order to assess how effectively Propeller service will likely perform in a real environment.

**Experimental Setup**.

We prototype Propeller on a 9-node Linux storage cluster to evaluate its scalability, where one node runs as Master Node, and the other 8 nodes run as Index Nodes. These nodes are connected by a NetGear ProSafe 24-port Gigabits switch. Each node in this cluster features an Intel Quad-Core Xeon X3440 (4 cores, 8M cache, 2.53GHz) CPU with $4 \sim 16$GB RAM running Ubuntu Linux Server 12.10. Each node is equipped with a Seagate Barracuda ST31000524AS 1TB, 7,200 RPM and 32MB Cache hard drive formatted as Ext4 for the experiments. We compare Propeller against the open-sourced relational database (MySQL) and Spotlight because they are the *de facto* standard file-search and/or file-metadata management solutions for most file systems [11], [15], [20], [30]. Furthermore, although the scalability of full-text search engines (e.g., ElasticSearch) and NoSQL databases (e.g., MongoDB) are significantly better than SQL databases, the indexing latency of them are expected comparable to SQL databases [48], because the essential data-structures (e.g., B+tree) used as index have similar time complexity (e.g., $O(logn)$ insert). Finally, the current SQL (MySQL cluster), NoSQL (MongoDB) and full text search (ElasticSearch) solutions can partition (shard) datasets based on a chosen key, and thus they are not aware of file-system access patterns. We leave their comparison to our future work.

To perform a fair comparison with a centralized MySQL, we run Propeller in the single-node mode (i.e., the Master Node and a single instance of Index Node run on the same Linux machine) to evaluate its single-node file-indexing and file-search performance. In this test, the MySQL data and Propeller index data are stored on the same clean Ext4 file system. Additionally, only B-tree based index is used in MySQL and Propeller tests. Propeller's update timeout is 5 seconds. MySQL's buffer size is set to 2GB, and the request batch size is 128 in both tests. Furthermore, we compare the file-search latency and accuracy of Propeller (in the single-node mode) against Spotlight on a Mac Mini machine with Intel i5-2415M CPU, 8GB RAM, 500GB, 5,400 RPM hard drive running Mac OSX 10.8.2. Finally, we also evaluate the I/O performance of Propeller by comparing it to several production-level Linux file systems.

### A. File Access-Causality Partitioning

We use three traditional applications to show the characteristics of file access-causality graphs. In order to capture the ACGs, we download the source code of three different real applications: Git version management software [3], Remote Procedure Call package Thrift [1] and Linux Kernel, and then compile them on the Propeller's FUSE-based file system on one client machine. The ACG obtained from compiling Thrift is drawn in Figure 7. The ACGs from other applications are similar. This graph clearly shows two disjoint connected components with no inter-partition accesses at all. This means that grouping the files corresponding to the connected components in the ACG graph would minimize inter-group accesses (in fact, to zero in this case). In the meantime, each connected component can be further divided into approximately equal-sized sub-graphs with the minimal inter-partition accesses (i.e., *balanced cut*) by applying graph partitioning algorithms [28], [37], [42]. Table II summarizes the key characteristics of the access-causality graphs obtained from the aforementioned three applications and the execution time of applying the METIS graph partitioning algorithm [28] on the largest connected component from each application.

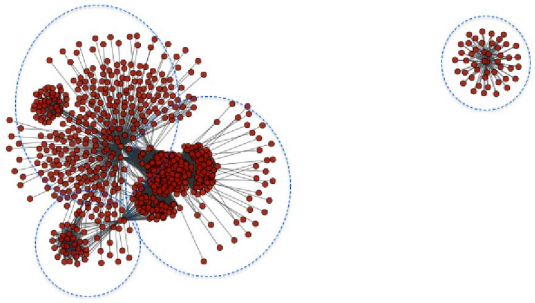


Fig. 7. **The Access-Causality Graph of Compiling Thrift**. Each vertex in the graph is a source file in the thrift application [1]. The blue cycles illustrates the potential cuts of this ACG. It clearly indicates that there are disconnected components in the access-causality graph for single application.

In conclusion, access-causality graph is an effective technique for clustering files in such a way that the inter-group accesses, the largest contributor to the file-indexing latency, can be significantly reduced or eliminated. Additionally, since splitting a large file-index partition (e.g., by running the METIS algorithm [28]) is a rare operation compared to file-indexing and file-search operations, and is performed in background, we argue that its relatively high overhead is acceptable in Propeller.

### B. Single-Node Performance

We compare Propeller against MySQL on a single Linux node to evaluate Propeller's performance advantages over the centralized file-search approaches [35], [36]. Two tables are used in MySQL in favor of its file-search performance: one for storing the full file path and inode attributes and the other for storing the mapping from keyword to file path, in which the keywords are extracted from the full file path. Due to the fact that publicly accessible file-system snapshots [6], [7] do not contain explicit file-access patterns necessary for the construction of access-causality partitions, we choose a set of well-known applications and open-source projects (e.g., Firefox, OpenOffice, Linux Kernel, etc.) to construct access-causality partitions, because they are representative of typical real-world workloads and are publicly accessible. To obtain a dataset of a desired scale, we duplicate these samples with an appropriate scaling factor.

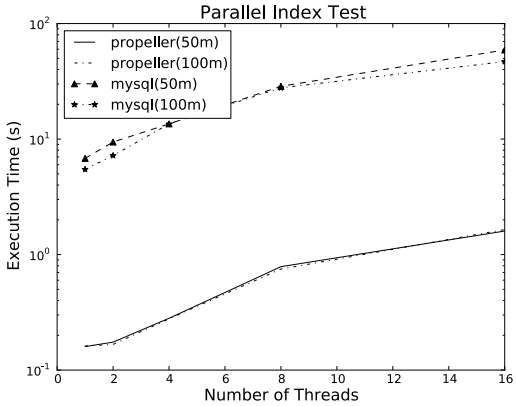



Fig. 8. **File Indexing Times (log)** on 50-million-file and 100-million-file datasets

**Scalable File-Indexing**. We start by feeding a sequence of concurrent file updates to both Propeller and MySQL on two different scaled datasets, one with 50-million files and the other with 100-million files. In this experiment, we create 1 through 16 processes to issue 10,000 update requests to Propeller and MySQL, respectively, and measure the execution times. It simulates the scenario that an application accesses a small fraction of the data compared to the whole system (i.e., 50/100-million files). In the Propeller experiment, each process issues IOs within one individual partition. In the MySQL experiment, each thread issues IOs to the same files accessed in the Propeller experiment. We have observed that the experimental results are consistent with different group sizes, thus we only present the results for the 1000-file-per-group experiment. As shown in Figure 8, the file-indexing performance of Propeller is $30 \sim 60$ times better than that of MySQL. Note that, in both data sets, the file-indexing performance of Propeller is similar, because all file-indexing IOs occur within a single group, so that the file update overhead is only determined by the size of the group. While in the MySQL case, it degrades significantly ($2\times$) from the 50-million-file dataset to the 100-million-file dataset, because the overhead is determined by the scale of the dataset. Thus, this experimental result indicates that the Propeller file-indexing performance is scalable. Furthermore, Propeller's performance degradation, as the number of threads increases, is due to the fact that the user-level Propeller threads issue parallel I/O requests to different files on the underlying Ext4 file system, resulting in mostly small and random IOs that are known to

| Application | # of Vertices (Files) | # of Edges | Total Weight of the Graph | Partitioning Time | Avg. # of Vertices of Resulting Partitions | Weight of Cut |
|---|---|---|---|---|---|---|
| Linux | 62331 | 5937685 | 6958560 | 35.37s | 30087/32244 | 92672 (1.33%) |
| Thrift | 775 | 8698 | 55454 | 0.042s | 359/369 | 316 (0.58%) |
| Git | 1018 | 2925 | 4162 | 0.018s | 494/524 | 1225 (29.4%) |

TABLE II
**EVALUATION OF THE FILE ACCESS-CAUSALITY PARTITIONING ALGORITHM (METIS [28])**. THE METIS ALGORITHM IS CAPABLE OF APPROXIMATELY DIVIDING THE ACCESS-CAUSALITY GRAPH INTO EQUAL-SCALE SUB-GRAPHS WHILE KEEPING THE CUT (I.E., INTER-PARTITION ACCESSES) MINIMAL. THE PERCENTAGE OF CUT IS THE SUM OF WEIGHTS OF THE EDGES CROSSING THE CUT DIVIDED BY TOTAL WEIGHT OF ALL EDGES.

perform very poorly on HDD-based storage system and form a performance bottleneck.

| Files (Million) | Propeller #1 | Propeller #2 | MySQL #1 | MySQL #2 |
|---|---|---|---|---|
| 10 | 0.099745 | 0.548982 | 5.60257 | 5.68406 |
| 20 | 0.758968 | 1.56552 | 12.7334 | 13.6765 |
| 30 | 1.05982 | 2.31851 | 18.9487 | 19.9276 |
| 40 | 1.19347 | 3.03695 | 25.1554 | 26.6886 |
| 50 | 1.6375 | 3.99506 | 32.4856 | 34.157 |

TABLE III
**GLOBAL FILE SEARCH (SECONDS)**: QUERY #1: SIZE $>$ 1 GB & MTIME $<$ 1DAY ; QUERY #2: KEYWORD "**FIREFOX**" & MTIME $<$ 1 WEEK.

We compare the file-search performance of Propeller and MySQL on the synthetically scaled-up namespaces. The namespaces are kept static in order to eliminate the impact of continuous file-index updates. We define two queries (listed in the caption of Table III) to evaluate the global-search performance of the two systems. The results, shown in Table III, indicate that these two queries in Propeller are on average 9.0 and 26.3 times faster than those in MySQL, respectively.

### C. Scalable Search Performance on Propeller Cluster

As described in Section IV, only the file-search requests involve multiple index nodes. Therefore, we evaluate the scalability of file-search API on the 9-node Propeller cluster. In this experiment, the number of Index Nodes scales from 1 to 8. After a fresh booting up, the same file-search requests are performed by Propeller on two different scales of datasets (50-million and 100-million files) in a close-loop manner. The latency of each request is measured. Each group node uses 16 threads to perform parallel searches on different groups located in the node. Within every cluster configuration, we issue the same file-search requests for 11 times. The "cold query" results are the measured search-latency values for the first queries of the 11-query sequences when the system is cold with no data cached, and the "warm query" results are the measured query-latency values averaged over the last 10 requests of the 11-query sequences.

The results shown in Table IV clearly indicate that the latency of file-search requests is significantly reduced linearly and even super-linearly as the Propeller cluster scales up, suggesting a high file-search scalability of Propeller in a distributed environment, especially when the cluster has more than 4 nodes. In the warm tests, the latencies improve super-linearly from $1 \rightarrow 4$ index nodes in the 100-million-file dataset

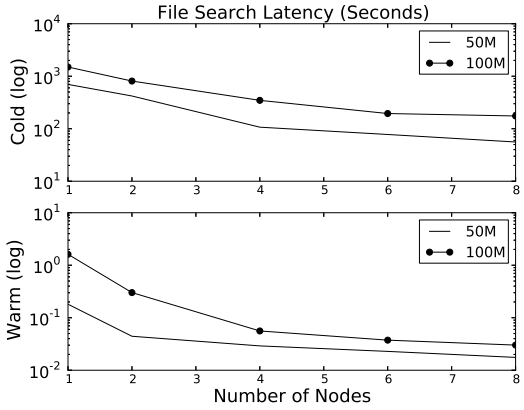



Fig. 9. **Propeller Cluster Search Performance** (log) on 50-million and 100-million files: "*finding the files larger than 16MB*"

| Test | Latency (seconds) | | | | |
|---|---|---|---|---|---|
| Number of Index Nodes | 1 | 2 | 4 | 6 | 8 |
| 100m (cold) | 1497.2 | 809.6 | 347.2 | 194.8 | 174.9 |
| 50m (cold) | 698.4 | 420.3 | 107.0 | 77.7 | 55.8 |
| 100m (warm) | 1.61 | 0.30 | 0.056 | 0.037 | 0.030 |
| 50m (warm) | 0.180 | 0.044 | 0.029 | 0.023 | 0.016 |

TABLE IV
**PROPELLER CLUSTER FILE-SEARCH LATENCY** (SECONDS) ON 50-MILLION AND 100-MILLION FILES: "*finding the files larger than 16MB*"

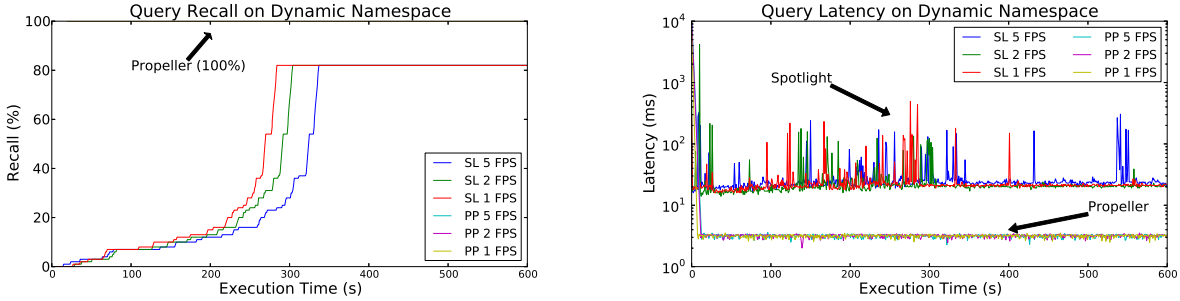
and $1 \rightarrow 2$ index nodes in the 50-million-file dataset. This is because that, with one or two nodes, the combined size of the file indices is larger than the size of the memory on each node, which causes frequent page faults when the file-search operations are performed. By distributing the groups among more nodes, each node's share of file indices is reduced proportionally to allow it to load the entire indices into its memory, avoiding page faults and resulting in much better performance. In summary, the reason for this great scalability of Propeller is that by distributing a large number of independent and small-scaled ACGs to different Index Nodes (see Section III), all Index Nodes are able to process the ACGs they house locally and in parallel, enabling Propeller to achieve very low file-search latency on very large datasets.

### D. Mixed Workloads

As described in Section IV, Propeller aggressively caches the indexing requests to effectively hide the indexing latency from the regular file IOs. However, this technique increases the latency of the search requests, because it must commit all modifications into the file indices before performing a file-

| Test | Dataset 1 | | | | Dataset 2 | | | |
|---|---|---|---|---|---|---|---|---|
| | Real(s) | User(s) | System(s) | Recall | Real(s) | User(s) | System(s) | Recall |
| Brute-Force (cold) | 51.878 | 0.469 | 5.676 | 100% | 110.372 | 1.974 | 20.178 | 100% |
| Spotlight (cold) | 2.755 | 0.022 | 0.043 | 60.6% | 3.605 | 0.020 | 0.102 | 13.86% |
| Propeller (cold) | 2.818 | 0.081 | 0.191 | 100% | 4.167 | 0.243 | 0.560 | 100% |
| Brute-Force (warm) | 5.185 | 0.259 | 3.414 | 100% | 90.561 | 1.990 | 21.383 | 100% |
| Spotlight (warm) | 0.021 | 0.011 | 0.007 | 60.6% | 0.068 | 0.012 | 0.007 | 13.86% |
| Propeller (warm) | 0.0015 | 0.0018 | 0.0014 | 100% | 0.0031 | 0.0045 | 0.0023 | 100% |

TABLE V
PERFORMANCE COMPARISON BETWEEN PROPELLER AND SPOTLIGHT ("*find files larger than* 16*MB*"). ALL THE EXPERIMENTS ARE MEASURED BY REAL TIME, USER TIME AND SYSTEM TIME. DATASET 1 HAS $138K$ FILES AND DATASET 2 HAS $487K$ FILES.




(a) Query Recall. Propeller's recall is 100%.

(b) Query Latency (log)

Fig. 11. **Query Accuracy and Latency on a Dynamic Namespace (Dataset 1)**. *SL*: Spotlight and *PP*: Propeller. After importing an Ubuntu snapshot ($89K$ files) into Dataset 1, we spawn a background I/O process to copy files into the dataset at various speeds. Then we continuously issue the query ("*find files larger than* 16*MB*") for 10 minutes to both Spotlight and Propeller and measure the query latency and accuracy.

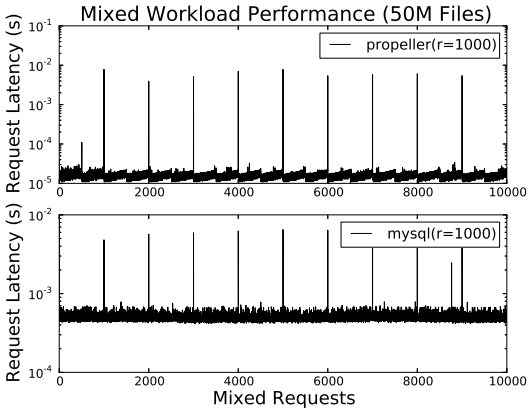



Fig. 10. **Mixed Workload (log)** (50 Million Files)

search request in order to guarantee the consistency of results. Thus, it is desirable to mix I/O operations with file-search requests in the I/O workloads to obtain a deeper understanding of the Propeller performance. To explicitly show the impact of file-search requests, we feed a synthetic workload consisting of 10,000 updates combined with file-attribute-search requests, to one group (1,000 files) on a 50-million-file dataset on both Propeller and MySQL, where there is one file-search request for every 1,024 updates. And the background re-indexing is triggered after every 500 updates to simulate the "timeout" effect in the lazy-indexing technique. As shown in Figure 10, the average latency of file re-indexing operations in Propeller ($15.6\mu s$) is $250\times$ lower than that in MySQL ($3,980.9\mu s$). This result proves that, with access-causality grouping, the performance penalty of synchronous-commit modifications before each file-search is very small in the Propeller solution due to the significantly reduced scale of an index, while in the MySQL solution, the update operations occur in the global namespace, which results in an extremely high latency. Furthermore, the file-index cache not only hides most of the re-indexing latency from the normal I/O operations but also reduces the number of modifications to be merged for the file-search requests due to the "background" merges triggered by the "timeout" mechanism). In summary, Propeller guarantees the consistency of file-search results with very little latency.

### *E. Performance Comparison against Spotlight*

Due to our lack of access to the Google Enterprise Search application [21], we use Spotlight [11], which is considered the most sophisticated desktop search engine and thus represents the state of the art in desktop search engines. We evaluate the efficiency and accuracy of Propeller by comparing the single-node Propeller prototype with Spotlight on a Mac Mini machine. The Spotlight index is completely rebuilt before each run of the Spotlight test, and the file system caches and disk caches are cleared before all experiments in this subsection. Due to the fact that the Propeller prototype lacks the rich set of file plug-ins to extract metadata from various file types that Spotlight has, we issue the same range query of the inode attributes to both Spotlight and Propeller. Additionally, we also perform a brute-forced search as the base-line of all experiments. We feed two datasets to Propeller to build the namespace: Dataset 1 ($138K$ files), the freshly installed image of Mac OSX 10.8.2 on Mac Mini, and Dataset 2 ($487K$ files), derived from Dataset 1 by combining it with

the file system snapshot of one author's Mac laptop. We compare Propeller against Spotlight in two aspects: *Static Namespace*, which represents the efficiency of file query, and *Dynamic Namespace*, which examines the impact of the crawling process.

*Static Namepace Test*. We repeatedly perform the same query 60 times with an interval of 1 second. The *cold query* results are measured for the first query, and the *warm query* results are the average values from the remaining 59 queries. The results illustrated in Table V show that Propeller is $2\% \sim 15\%$ slower than Spotlight for cold queries in the cold-cache test, but it is $14 \sim 21.94$ times faster than Spotlight for warm queries in the warm-cache test. We argue that the warm-cache performance is more important than the cold-cache one, as the file-search results will most likely be accessed frequently and repeatedly by the parallel executed analytics applications on different client nodes.

*Dynamic Namespace Test*. In this test, we first import a Linux virtual machine (Ubuntu) snapshot into the dataset. Then we immediately spawn a background I/O process and start a foreground process to continuously search files, as described in Figure 11. Figure 11(a) shows that the search accuracy metric, recall (see Section II), of Spotlight reaches the maximum value of $(82.0\%)$ at different speeds, which is determined by the background I/O intensity. Figure 11(b) shows that the average query latency of Propeller $(3.1ms)$ is 9 times faster than Spotlight $(28.5ms)$. Due to space limit, we only present the results for Dataset 1, since the results for Dataset 2 are similar. The results clearly demonstrate that Propeller is superior to Spotlight in dynamic namespace performance on both query latency and accuracy, which are of significant importance when the file-search API is integrated into big-data applications.

It is also noteworthy that the inode attribute index in the Propeller prototyping process is implemented in a serialized KD-tree. It means that, for each index group, Propeller has to load the entire KD-tree in RAM, which accounts for the most of its latency, as indicated by $time_{real} - (time_{user} + time_{system})$ in the cold-cache tests. With a specialized design of the on-disk structure of KD-tree, which is left to our future work, it is possible to substantially reduce the IOs so that the query latency of Propeller can be dramatically improved further. In summary, the advantages of access-causality grouping in Propeller enable it to provide real-time file search service, which is infeasible to state-of-the-art file-search engines.

### F. Raw I/O Performance

We evaluate Propeller's raw I/O performance to assess the inline file-indexing overhead. We run the PostMark benchmark [29] on several file systems that are categorized into two types: native (Ext4/Btrfs) and FUSE-based (NTFS/ZFS/Propeller) file systems on a single Linux machine. Additionally, we implement a pass-through FUSE file system (PTFS) that passes through I/O requests to the underlying Ext4 file system in order to measure the overhead introduced by FUSE. The PostMark benchmark creates $50000$ files under 200 subdirectories within each file system. The results, shown in Table VI, indicate that Propeller is about $2.37\times$ slower than the FUSE pass-through implementation. The reason is because Propeller does inline indexing for the corresponding file.

| FS | Files Created per second | Read/Write Throughput | Real/User/Sys Time (s) |
|---|---|---|---|
| Ext4 | 16747 | 391KB/84MB | 5.44/0.22/1.92 |
| Btrfs | 5582 | 130KB/28.1MB | 7.85/0.37/7.44 |
| PTFS | 6289 | 146.76KB/31.51MB | 8.02/1.63/5.74 |
| NTFS-3g | 2392 | 55.9KB/12MB | 12.5/4.80/5.09 |
| ZFS-fuse | 2093 | 58.71KB/12.61MB | 20.4/8.95/6.14 |
| Propeller | 2644 | 61.79KB/12.61MB | 68.1/11.5/12.1 |

TABLE VI
**POSTMARK BENCHMARK RESULTS**. WE COMPARE PROPELLER AGAINST NATIVE FILE SYSTEMS (EXT4/BTRFS) AND TWO FUSE-BASED FILE SYSTEMS (NTFS-3G, ZFS-FUSE). WE ALSO COMPARE IT AGAINST PTFS, A PASS-THROUGH FUSE FILE SYSTEM, TO EVALUATE THE OVERHEAD INTRODUCED BY FUSE. PROPELLER HAS A COMPARABLE RAW I/O PERFORMANCE TO OTHER FUSE-BASED FILE SYSTEMS SUCH AS NTFS-3G AND ZFS-FUSE.

In summary, the FUSE-based Propeller prototype, with its advanced file-search functionality and high query-accuracy guarantee, has overheads that are comparable to other FUSE-based advanced file systems (e.g., NTFS-3g and ZFS-fuse) that also offer more functionalities (e.g., volume management and end-to-end integrity) than Ext4. Additionally, in a cluster environment, the file-indexing overhead shifts to the distributed index nodes, which is substantially amortized by the relatively high network overhead.

## VI. CONCLUSION AND FUTURE WORK

This paper presents Propeller, a distributed real-time file-search service. By applying a novel file-clustering mechanism, called *Access-Causality* partitioning, and several other optimization techniques, Propeller offers an inline file-indexing capability with reasonable raw I/O performance. The evaluations show that Propeller outperforms a centralized solution (MySQL) by 2$\sim$3 orders of magnitude in the file-indexing and file-search performance, and has much higher accuracy and substantially lower query latency than the state-of-the-art desktop search engine (Spotlight). The cluster implementation of Propeller also demonstrates its almost linear file-search latency scalability.


## ACKNOWLEDGEMENTS

This work is supported by the US NSF under Grant No. NSF-CNS-1116606, NSF-CNS- 1016609, NSF-IIS-0916859, and NSF-CCF-0937993. The authors are also grateful to anonymous reviewers and our shepherd, Dr. Michael E. Factor and Professor Andre Brinkmann, for their feedback and guidance.


## REFERENCES


[1] Apache thrift. http://thrift.apache.org/.
[2] Filesystem in userspace. http://fuse.sourceforge.net/.
[3] Git: a free and open source distributed version control system. http://git-scm.com/.

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

[5] Precision and recall. http://en.wikipedia.org/wiki/Precision_and_recall.

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

[7] N. Agrawal and et al. Generating realistic impressions for file-system benchmarking. In *FAST '09*, 2009.

[8] R. Ananthanarayanan and et al. Photon: Fault-tolerant and scalable joining of continuous data streams. SIGMOD '13.

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

[10] Apple Inc. File system events programming guide.

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

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

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

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

[15] H. Company. HP storeall storage.

[16] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. OSDI'04.

[17] J. Fernandez-Sanguino. The debian gnu/linux faq chapter 8 - the debian package management tools. http://www.debian.org/doc/FAQ/ch-pkgtools.en.html.

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

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

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

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

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

[23] J. Gray and et al. Scientific data management in the coming decade. *SIGMOD Rec.*, 34(4), 2005.

[24] L. Hagen and A. Kahng. New spectral methods for ratio cut partitioning and clustering. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 11(9):1074–1085, 1992.

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

[26] 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.

[27] 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, Microsoft Research, 2006.

[28] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.

[29] J. Katcher. Postmark: A new file system benchmark. *System*, (3022), 1997.

[30] 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*.

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

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

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

[34] J. McCutchan, R. Love, and A. Griffis. Inotify - get your file system supervised. http://inotify.aiken.cz/.

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

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

[37] A. Y. Ng, M. I. Jordan, and Y. Weiss. On spectral clustering: Analysis and an algorithm. In *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS*, pages 849–856. MIT Press, 2001.

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

[39] 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.

[40] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI '10*, pages 4–6, 2010.

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

[42] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. KDD '12.

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

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

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

[46] 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.

[47] 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.

[48] L. Xu, Z. Huang, H. Jiang, L. Tian, and D. Swanson. Providing flexible file-level data filtering for big data analytics. Technical Report 131, University of Nebraska-Lincoln, Apr. 2013.