Just-in-Time Analytics on Large File Systems

H. Howie Huang, Nan Zhang, Wei Wang, Gautam Das, and Alexander S. Szalay

Abstract—As file systems reach the petabytes scale, users and administrators are increasingly interested in acquiring high-level analytical information for file management and analysis. Two particularly important tasks are the processing of aggregate and top-k queries which, unfortunately, cannot be quickly answered by hierarchical file systems such as ext3 and NTFS. Existing preprocessing-based solutions, e.g., file system crawling and index building, consume a significant amount of time and space (for generating and maintaining the indexes) which in many cases cannot be justified by the infrequent usage of such solutions. In this paper, we advocate that user interests can often be sufficiently satisfied by *approximate*—i.e., statistically accurate—answers. We develop *Glance*, a just-in-time sampling-based system which, after consuming a small number of disk accesses, is capable of producing extremely accurate answers for a broad class of aggregate and top-k queries over a file system without the requirement of any prior knowledge. We use a number of real-world file systems to demonstrate the efficiency, accuracy, and scalability of Glance.

Index Terms—Data analytics, file systems

1 Introduction

Today, a file system with billions of files, millions of directories, and petabytes of storage is no longer an exception [31]. As file systems grow, users and administrators are increasingly keen to perform complex queries [39], [49], such as "How many files have been updated since 10 days?" and "Which are the top five largest files that belong to John?" The first is an example of *aggregate queries* which provide a high-level summary of all or part of the file system, while the second is *top-k queries* which locate the k files and/or directories that have the highest score according to a scoring function. Fast processing of aggregate and top-k queries is often needed by applications that require *just-in-time analytics* over large file systems, such as data management, archiving, etc. The just-in-time requirement is defined by two properties: 1) file-system analytics must be completed with a small *access cost*—i.e., after accessing only a small percentage of directories/files in the system (in order to ensure efficiency), and 2) the analyzer holds no prior knowledge (e.g., preprocessing results) of the file system being analyzed. For example, in order for a librarian to determine how to build an image archive from an external storage media (e.g., a Blue-ray disc), he/she may have to first estimate the total size of picture files stored on the external media—the librarian needs to complete data analytics quickly, over an alien file system that has never been seen before.

Unfortunately, hierarchical file systems (e.g., ext3 and NTFS) are not well equipped for the task of just-in-time analytics [45]. The deficiency is in general due to the lack of a *global view* (i.e., high-level statistics) of metadata information (e.g., size, creation, access, and modification time). For efficiency concerns, a hierarchical file system is usually designed to limit the update of metadata information to individual files and/or the immediately preceding directories, leading to localized views. For example, while the last modification time of an individual file is easily retrievable, the last modification time of files that belong to user John is difficult to obtain because such metadata information is not available at the global level.

Currently, there are two approaches for generating high-level statistics from a hierarchical file system, and thereby answering aggregate and top-k queries: 1) The first approach is to scan the file system upon the arrival of each query, e.g., the *find* command in Linux, which is inefficient for large file systems. While storage capacity increases $\sim$60 percent per year, storage throughput and latency have much slower improvements; thus, the amount of time required to scan an off-the-shelf hard drive or external storage media has increased significantly over time to become infeasible for just-in-time analytics. The above-mentioned image-archiving application is a typical example, as it is usually impossible to completely scan an alien Blue-ray disc efficiently. 2) The second approach is to utilize prebuilt indexes which are regularly updated [3], [7], [26], [34], [38], [42]. Many desktop search products belong to this category—e.g., Google Desktop [23] and Beagle [5]. While this approach is capable of fast query processing once the (slow) index building process is complete, it may not be suitable or applicable to many just-in-time applications:

- *H.H. Huang is with the Department of Electrical and Computer Engineering, School of Engineering and Applied Science, The George Washington University, 801 22nd Street NW, Washington, DC 20052. E-mail: howie@gwu.edu.*
- *N. Zhang is with the Department of Computer Science, The George Washington University, 801 22nd Street NW, Washington, DC 20052. E-mail: nzhang10@gwu.edu.*
- *W. Wang is with the Department of Computer and Information Science, University of Delaware, 101 Smith Hall, Newark, DE 19716. E-mail: weiwang@udel.edu.*
- *G. Das is with the Computer Science and Engineering Department, University of Texas at Arlington, 500 UTA Blvd., 626, Engineering Research Building, Arlington, TX 76010. E-mail: gdas@uta.edu.*
- *A.S. Szalay is with the Department of Physics and Astronomy, The Johns Hopkins University, 3701 San Martin Drive, Baltimore, MD 21218. E-mail: szalay@jhu.edu.*

Manuscript received 7 Mar. 2011; revised 26 June 2011; accepted 3 Sept. 2011; published online 30 Sept. 2011.
Recommended for acceptance by D. Talia.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2011-03-0153.
Digital Object Identifier no. 10.1109/TC.2011.186.

- Index building can be unrealistic for many applications that require just-in-time analytics over an alien file system.
- Even if index can be built upfront, its significant cost may not be justifiable if the index is not frequently used afterward. Unfortunately, this is common for some large file systems, e.g., storage archives or scratch data for scientific applications scarcely require the global search function offered by the index, and may only need analytical queries to be answered infrequently (e.g., once every few days). In this case, building and updating an index is often an overkill given the high amortized cost.
- There are also other limitations of maintaining an index. For example, prior work [48] has shown that even after a file has been completely removed (from both the file system and the index), the (former) existence of this file can still be inferred from the index structure. Thus, a file system owner may choose to avoid building an index for privacy concerns.

To enable just-in-time analytics, one must be able to perform an on-the-fly processing of analytical queries, over traditional file systems that normally have insufficient metadata to support such complex queries. We achieve this goal by striking a balance between query answer accuracy and cost—providing approximate (i.e., statistically accurate) answers which, with a high confidence level, reside within a close distance from the precise answer. For example, when a user wants to count the number of files in a directory (and all of its subdirectories), an approximate answer of 105,000 or 95,000, compared with the real answer of 100,000, makes little difference to the high-level knowledge desired by the user. The higher cost a user is willing to pay for answering a query, more accurate the answer can be.

To this end, we design and develop *Glance*, a just-in-time query processing system which produces accurate query answers based on a small number of samples (files or folders) that can be collected from a very large file system with a few disk accesses. Glance is file-system agnostic, i.e., it can be applied instantly over any new file system and work seamlessly with the tree structure of the system. Glance removes the need of disk crawling and index building, providing just-in-time analytics without a priori knowledge or preprocessing of the file systems. This is desirable in situations when the metadata indexes are not available, a query is not supported by the index, or query processing is only scarcely needed.

Using sampling for processing analytical queries is by no means new. Studies on sampling flat files, hashed files, and files generated by a relational database system (e.g., a B+-tree file) started more than 20 years ago—see survey [41]—and were followed by a myriad of work on database sampling for approximate query processing in decision support systems—see tutorials [4], [15], [22]. A wide variety of sampling techniques, e.g., simple random sampling [40], stratified [10], reservoir [50], and cluster sampling [11], have been used. Nonetheless, to the best of our knowledge, there has been no existing work on using sampling to support efficient aggregate and top-k query processing over a large hierarchical file system, i.e., one with numerous files organized in a complex folder structure (tree-like or directed acyclic graph (DAG)).

Our main contributions are twofold: 1) Glance consists of two algorithms, *FS_Agg* and *FS_TopK*, for the approximate processing of aggregate and top-k queries, respectively. For just-in-time analytics over very large file systems, we develop a random descent technique for unbiased aggregate estimations and a pruning-based technique for top-k query processing. 2) We study the specific characteristics of real-world file systems and derive the corresponding enhancements to our proposed techniques. In particular, according to the distribution of files in real-world file systems, we propose a high-level crawling technique to significantly reduce the error of query processing. Based on an analysis of accuracy and efficiency for the descent process, we propose a breadth-first implementation to reduce both error and overhead. We evaluate Glance over both real-world (e.g., NTFS, NFS, Plan 9) and synthetic file systems and find very promising results—e.g., 90 percent accuracy at 20 percent cost. Furthermore, we demonstrate that Glance is scalable to one billion of files and millions of directories.

We would like to note, however, that Glance also has its limitations—there are certain ill-formed file systems that malicious users could potentially construct that Glance cannot effectively handle. While we plan to address security applications in future work, our argument of Glance being a practical system for just-in-time analytics is based upon the fact that these systems rarely exist in practice. For example, Glance cannot accurately answer aggregate queries if a large number of folders are hundreds of levels below root. Nonetheless, real-world file systems would have far smaller depth, making such a scenario unlikely to occur. Similarly, Glance cannot efficiently handle cases where all files have extremely close scores. This, however, is contradicted by the heavy-tailed distribution observed on most metadata attributes in real-world file systems [2].

The rest of the paper is organized as follows: Section 2 presents the problem definition. We introduce the Glance system architecture in Section 3. In Sections 4 and 5, we describe FS_Agg and FS_TopK for processing aggregate and top-k queries, respectively. The evaluation results are shown in Section 6. Section 7 reviews the related work, followed by the discussion in Section 8. We conclude in Section 9.

2 Problem Statement

We now define the analytical queries, i.e., aggregate and top-k ones, which we focus on in this paper. The examples we list below will be used in the experimental evaluation for testing the performance of Glance.

2.1 Aggregate Queries

In general, aggregate queries are of the form *SELECT AGGR(T) FROM D WHERE Selection Condition*, where D is a file system or storage device, T is the target piece of information, which may be a metadata attribute (e.g., size, time stamp) of a file or a directory, *AGGR* is the aggregate function (e.g., COUNT, SUM, AVG), and *Selection Condition* specifies which files and/or directories are of interest. First, consider a system administrator who is interested in the total number of files in the system. In this case, the

aggregate query that the administrator would like to issue can be expressed as:

Q1: SELECT COUNT(files) FROM filesystem;

The operating system provides tools (e.g., Linux's find and Window's dir) to gather statistics for answering such a query. However, because each directory only maintains its local metadata, the operating system has to recursively scan the metadata portion of the file system (e.g., the Master File Table for NTFS), which may lead to a longer running time. Further, the administrator may be interested in knowing the total size of various types of document files, e.g.,

Q2: SELECT SUM(file.size) FROM filesystem WHERE file.extension IN {"txt," "doc"};

If the administrator wants to compute the average size of all exe files from user John, the query becomes

Q3: SELECT AVG(file.size) FROM filesystem WHERE file.extension = "exe" AND file.owner = "John";

Aggregate queries can also be more complex—the following example shows a nested aggregate query for scientific computing applications. Suppose that each directory is corresponding to a sensor and contains a number of files corresponding to the sensor readings received at different time. A physicist may want to count the number of sensors that has received at least one reading during the last 12 hours, i.e.,

*Q4: SELECT COUNT(directories) FROM filesystem WHERE EXISTS (SELECT * FROM filesystem WHERE file.dirname = directory.name AND file.mtime BETWEEN (now − 12 hours) AND now).*

2.2 Top-k Queries

In this paper, we also consider top-k queries of the form *SELECT TOP k FROM D WHERE Selection Condition ORDER BY T DESCENDING/ASCENDING*, where T is the *scoring function* based on which the top-k files or directories are selected. For example, a system administrator may want to select the 100 largest files, i.e.,

Q5: SELECT TOP 100 files FROM filesystem ORDER BY file.size DESCENDING;

Another example is to find the 10 most recently created directories that were modified yesterday, i.e.,

Q6: SELECT TOP 10 directories FROM filesystem WHERE directory.mtime BETWEEN (now − 24 hours) AND now ORDER BY directory.ctime DESCENDING;

We note that, to approximately answer a top-k query, one shall return a list of k items that share a large percentage of common ones with the precise top-k list.

Current operating systems and storage devices do not provide APIs which directly support the above-defined aggregate and top-k queries. The objective of just-in-time analytics can be stated as follows.

2.3 Problem Statement

The objective of Just-In-Time Analytics over File Systems is to enable the efficient approximate processing of aggregate and top-k queries over a file system by using the file/directory access APIs provided by the operating system.

To complete the problem statement, we need to determine how to measure the efficiency and accuracy of query processing. For the purpose of this paper, we measure the query efficiency in two metrics: 1) *query time*, i.e., the runtime of query processing, and 2) *query cost*, i.e., the ratio of the number of directories visited by Glance to that of crawling the file system (i.e., the total number of directories in the system). We assume that one disk access is required for reading a new directory.[1] Thus, the query cost approximates the number of disk accesses required by Glance. The two metrics, query time and cost, are positively correlated—the higher the query cost, the more directories the algorithm has to sample, leading to a longer runtime.

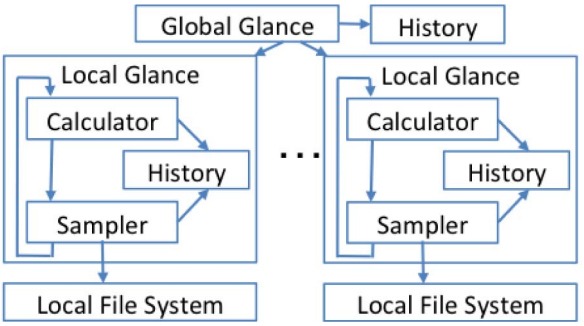


Fig. 1. Glance architecture.

While the efficiency measures are generic to both aggregate and top-k query processing, the measures for query accuracy are different. For aggregate queries, we define the query accuracy as the relative error of the approximate answer apx compared with the precise one ans—i.e., $|apx - ans|/|ans|$. For top-k queries, we define the accuracy as the percentage of items that are common in the approximate and precise top-k lists. The accuracy level required for approximate query processing depends on the intended application.

3 System Architecture of Glance

We envision Glance to have the following three properties: 1) Scalability—Glance should work on file systems consisting of thousands to billions of files. It should also support local file systems as well as distributed ones which cover thousands of machines in a large organization. 2) Accuracy—Glance should provide high accuracy for a large class of aggregate and top-k queries. 3) Efficiency—Glance should minimize both the number of disk accesses and the runtime. To this end, Glance features a two-level architecture which is depicted in Fig. 1. The lower level consists of local components running independently to provide just-in-time analytics over the local file system. The upper level is a global component which manages the execution of local components to produce global query answers.

3.1 Local Components

Each local component of Glance consists of three modules: a *sampler* which draws samples from the local file system, a *calculator* which estimates local query answers based on the samples, and a *history* module which saves the historically

1. Admittedly, this is a simplified assumption as certain file systems such as NTFS require multiple disk accesses for reading one directory. Nonetheless, we would like to note that this simplified assumption allows us to perform theoretical analysis on the performance (i.e., accuracy versus access cost) of our proposed algorithms without knowing the underlying distribution of files in the system.

issued queries and their answers. The main process of (approximate) query processing is handled by the sampler and the calculator. Note that both modules can be launched in an iterative fashion to achieve a desired level of trade-off between accuracy and cost. To determine whether to collect more samples to refine the estimation, the user-determined stopping condition can be specified on either accuracy or cost—e.g., an upper bound on the runtime or the estimation variance. If an upper bound on the estimation variance is specified, we can approximate the variance with the sample variance after finite-population correction [14]. The history module is designed to reduce the cost of sampling. It stores the previous *estimations* (generated by the sampler and the calculator) over parts (e.g., subtrees) of the file system. The idea is that the sampler (in future executions) can leverage the history to limit the search space to the previously unexplored part of the file system, unless it determines that the history is obsolete (e.g., according to a predefined validity period). Note that the history is continuously updated by the sampler and the calculator to include newly discovered directories and to update the existing estimations.

3.2 Global Processing for Distributed Systems

To support query processing over a distributed file system, Glance includes a global component which manages the local components for individual file systems. Here, we assume that there exists a directory service which can be utilized by Glance to communicate with the distributed machines. Note that to answer a global query, it is not necessary for Glance to sample every local file system, especially when there is a large number of them. A simple approach is for the global component to choose a small number of file systems uniformly at random and then run the local components over these selected systems. Based on the local results, the global calculator can estimate an aggregate result for the global system.

We note that the local component on each local file system works independently, requiring neither synchronization nor communication (with each other). This makes our Glance system highly scalable to a distributed system. For example, in the event where a local system becomes unavailable, busy, or takes a considerable amount of time for response, another machine can be selected as a replacement, making Glance resilient to system and network faults, which are the norm in a distributed environment.

4 Aggregate Query Processing

In this section, we develop *FS_Agg*, our algorithm for processing aggregate queries. We first describe *FS_Agg_Basic*, a vanilla algorithm which illustrates our main idea of aggregate estimation without bias through a random descent process within a file system. Then, we describe two ideas to make the vanilla algorithm practical over very large file systems: *high-level crawling* leverages the special properties of a file system to reduce the standard error of estimation, and *breadth-first implementation* improves both accuracy and efficiency of query processing. Finally, we combine all three techniques to produce FS_Agg.

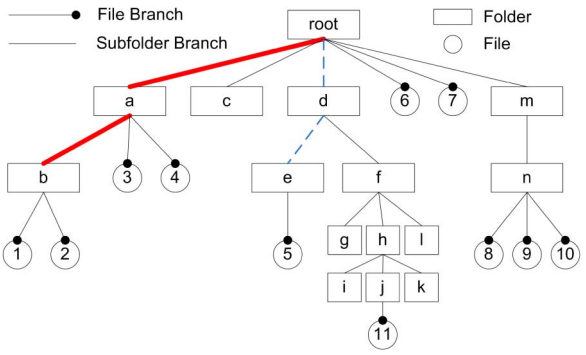


Fig. 2. Random descents on a tree-like structure.

4.1 FS_Agg_Basic

4.1.1 A Random Descent Process

In general, the folder organization of a file system can be considered as a tree or a directed acyclic graph, depending on whether the file system allows hard links to the same file. The random descent process we are about to discuss can be applied to both cases with little change. For the ease of understanding, we first focus on the case of tree-like folder structure, and then discuss a simple extension to DAG at the end of this section.

Fig. 2 depicts a tree structure with root corresponding to the root directory of a file system, which we shall use as a running example throughout the paper. One can see from the figure that there are two types of nodes in the tree: folders (directories) and files. A file is always a leaf node. The children of a folder consist of all subfolders and files in the folder. We refer to the branches coming out of a folder node as subfolder branches and file branches, respectively, according to their destination type. We refer to a folder with no subfolder branches as a *leaf folder*. Note that this differs from a leaf in the tree, which can be either a file or a folder containing neither subfolder nor file. The random descent process starts from the root and ends at a leaf folder. At each node, we choose a subfolder branch of the node uniformly at random for further exploration. During the descent process, we evaluate all file branches encountered at each node along the path, and generate an aggregate estimation based on these file branches.

To make the idea more concrete, consider an example of estimating the COUNT of all files in the system. At the beginning of random descent, we access the root to obtain the number of its file and subfolder branches f_0 and s_0, respectively, and record them as our evaluation for the root. Then, we randomly choose a subfolder branch for further descent, and repeat this process until we arrive at a folder with no subfolder. Suppose that the descent process continues for h ($h \geq 1$) steps, and the numbers we recorded for the ith step ($i \in [1, h]$) are f_i and s_i, for the number of file and subfolder branches, respectively. Note that $s_h = 0$ because each descent ends at a leaf folder. We estimate the COUNT of all files as

$$\tilde{n} = \sum_{i=0}^{h} \left(f_i \cdot \prod_{j=0}^{i-1} s_j \right), \tag{1}$$

where $\prod_{j=0}^{i-1} s_j$ is assumed to be 1 when $i = 0$. Two examples of such a random descent process are marked in Fig. 2 as red solid and blue dotted lines, respectively. The solid descent produces $\langle f_0, f_1, f_2\rangle = \langle 2, 2, 2\rangle$ and $\langle s_0, s_1, s_2\rangle = \langle 4, 1, 0\rangle$, leading to an estimation of $2 + 8 + 8 = 18$. The dotted one produces $\langle f_0, f_1, f_2\rangle = \langle 2, 0, 1\rangle$ and $\langle s_0, s_1, s_2\rangle = \langle 4, 2, 0\rangle$, leading to an estimation of $2 + 0 + 8 = 10$. The random descent process can be repeated multiple times (by restarting from the root) to produce a more accurate result (by taking the average of estimations generated by all descents).

4.1.2 Unbiasedness

Somewhat surprisingly, the estimation produced by each random descent process is completely *unbiased*—i.e., the expected value of the estimation is exactly equal to the total number of files in the system. To understand why, consider the total number of files at the ith level (with root being Level 0) of the tree (e.g., Files 1 and 2 in Fig. 2 are at Level 3), denoted by F_i. According to the definition of a tree, each i-level file belongs to one and only one folder at Level $i - 1$. For each $(i - 1)$-level folder v_{i-1}, let $|v_{i-1}|$ and $p(v_{i-1})$ be the number of (i-level) files in v_{i-1} and the probability for v_{i-1} to be reached in the random descent process, respectively. One can see that $|v_{i-1}|/p(v_{i-1})$ is an unbiased estimation for $F(i)$ because

$$E\left(\frac{|v_{i-1}|}{p(v_{i-1})}\right) = \sum_{v_{i-1}} \left(p(v_{i-1}) \cdot \frac{|v_{i-1}|}{p(v_{i-1})}\right) = F_i. \tag{2}$$

With our design of the random descent process, the probability $p(v_{i-1})$ is

$$p(v_{i-1}) = \prod_{j=0}^{i-2} \frac{1}{s_j(v_{i-1})}, \tag{3}$$

where $s_j(v_{i-1})$ is the number of subfolder branches for each node encountered on the path from the root to v_{i-1}. Our estimation in (1) is essentially the sum of the unbiased estimations in (2) for all $i \in [1, m]$, where m is the maximum depth of a file. Thus, the estimation generated by the random descent is unbiased.

The unbiasedness of our random decent process completely eliminates a major component of estimation error (note that the mean square error of estimation, MSE, is the sum of bias^2 and variance of estimation). We shall focus on reducing the other component of estimation error, i.e., estimation variance, in the latter part of the paper.

4.1.3 Processing of Aggregate Queries

While the above example is for estimating the COUNT of all files, the same random descent process can be used to process queries with other aggregate functions (e.g., SUM, AVG), with selection conditions (e.g., COUNT all files with extension ".JPG"), and in file systems with a DAG instead of tree structure. We now discuss these extensions. In particular, we shall show the only change required for all these extensions is on the computation of f_i.

4.1.4 SUM

For the COUNT query, we set f_i to the number of files in a folder. To process a SUM query over a file metadata attribute (e.g., file size), we simply set f_i as the SUM of such an attribute over all files in the folder (e.g., total size of all files). In the running example, consider the estimation of SUM of numbers shown on all files in Fig. 2. The solid and dotted random walks will return $\langle f_0, f_1, f_2\rangle = \langle 15, 7, 3\rangle$ and $\langle 15, 0, 5\rangle$, respectively, leading to the same estimation of 55. The unbiasedness of such an estimation follows in analogy from the COUNT case.

4.1.5 AVG

A simple way to process an AVG query is to estimate the corresponding SUM and COUNT, respectively, and compute AVG as SUM/COUNT. Note, however, that such an estimation is no longer unbiased, because the division of two unbiased estimations is not necessarily unbiased. While an unbiased AVG estimation may be desired for certain applications, we have proved a negative result that it is impossible to answer an AVG query without bias unless one accesses the file system for almost as many as times as *crawling* the file system. We omit the detailed proof here due to the space limitation. Nonetheless, for practical purposes, estimating AVG as SUM/COUNT is in general very accurate, as we shall show in the experimental results.

4.1.6 Selection Conditions

To process a query with selection conditions, the only change required is, again, on the computation of f_i. Instead of evaluating f_i over all file branches of a folder, to answer a conditional query, we only evaluate f_i over the files that satisfy the selection conditions. For example, to answer a query SELECT COUNT(*) FROM Files WHERE file.extension = "JPG," we should set f_i as the number of files under the current folder with extension JPG. Similarly, to answer "SUM($file\_size$) WHERE owner = John," we should set f_i to the SUM of sizes for all files (under the current folder) which belong to John. Due to the computation of f_i for conditional queries, the descent process may be *terminated early* to further reduce the cost of sampling. Again, consider the query condition of (owner = John). If the random descent reaches a folder which cannot be accessed by John, then it has to terminate immediately because of the lack of information for further descent.

4.1.7 Extension to DAG Structure

Finally, for a file system featuring a DAG (instead of tree) structure, we again only need to change the computation of f_i. Almost all DAG-enabled file systems (e.g., ext2, ext3, and NTFS) provide a *reference count* for each file which indicates the number of links in the DAG that point to the file.[2] For a file with r links, if we use the original algorithm discussed above, then the file will be counted r times in the estimation. Thus, we should discount its impact on each estimation with a factor of $1/r$. For example, if the query being processed is the COUNT of all files, then we should compute $f_i = \sum_{f \in F}(1/r(f))$, where F is the set of files under the current folder, and $r(f)$ is the number of links to each file f. Similarly, to estimate the SUM of all file sizes, we should compute $f_i = \sum_{f \in F}(size(f)/r(f))$, where $size(f)$ is

2. In ext2 and ext3, for example, the system provides the number of hard links for each file. Note that for soft links, we can simply ignore them during the descent process. Thus, they bear no impact on the final estimation.

the file size of file f. One can see that with this discount factor, we maintain an unbiased estimation over a DAG file system structure.

4.2 Disadvantages of FS_Agg_Basic

While the estimations generated by FS_Agg_Basic are unbiased for SUM and COUNT queries, it is important to understand that the error of an estimation comes from not only bias but also variance (i.e., standard error). A problem of FS_Agg_Basic is that it may produce a high estimation variance for file systems with an undesired distribution of files, as illustrated by the following theorem:

Theorem 1. *The variance of estimation produced by a random descent on the number of h-level files F_h is*

$$\sigma(h)^2 = \left(\sum_{v \in L_{h-1}} \left(|v|^2 \cdot \prod_{j=0}^{h-2} s_j(v)\right)\right) - F_h^2, \tag{4}$$

where L_{h-1} is the set of all folders at Level $h-1$, $|v|$ is the number of files in a folder v, and $s_j(v)$ is the number of subfolders for the Level-j node on the path from the root to v.

Proof. Consider an $(h-1)$-level folder v. If the random descent reaches v, then the estimation it produces for the number of h-level files is $|v|/p(v)$, where $p(v)$ is the probability for the random descent to reach v. Let $\delta(h)$ be the probability that a random descent terminates early before reaching a Level-$(h-1)$ folder. Since each random descent reaches at most one Level-$(h-1)$ folder, the estimation variance for F_h is

$$\sigma(h)^2 = \delta(h) \cdot F_h^2 + \sum_{v \in L_{h-1}} p(v) \cdot \left(\frac{|v|}{p(v)} - F_h\right)^2 \tag{5}$$

$$= \delta(h) \cdot F_h^2 + \sum_{v \in L_{h-1}} \left(\frac{|v|^2}{p(v)} - 2|v|F_h + p(v) \cdot F_h^2\right) \tag{6}$$

$$= \left(\sum_{v \in L_{h-1}} \frac{|v|^2}{p(v)}\right) - F_h^2. \tag{7}$$

Since $p(v) = 1/\prod_{j=0}^{h-2} s_j(v)$, the theorem is proved. □

One can see from the theorem that the existence of two types of folders may lead to an extremely high estimation variance: one type is *high-level leaf folders* (i.e., "shallow" folders with no subfolders). Folder c in Fig. 2 is an example. To understand why such folders lead to a high variance, consider (7) in the proof of Theorem 1. Note that for a large h, a high-level leaf folder (above Level-$(h-1)$) reduces $\sum_{v \in L_{h-1}} p(v)$ because once a random descent reaches such a folder, it will not continue to retrieve any file in Level-h (e.g., Folder c in Fig. 2 stops further descents for $h = 3$ *or* 4). As a result, the first item in (7) becomes higher, increasing the estimation variance. For example, after removing Folder c from Fig. 2, the estimation variance for the number of files on Level 3 can be reduced from 24 to 9.

The other type of "ill-conditioned" folders are those *deep-level folders* which reside at much lower levels than others (i.e., with an extremely large h). An example is Folder j in Fig. 2. The key problem arising from such a folder is that the probability for it to be selected is usually extremely small, leading to an estimation much larger than the real value if the folder happens to be selected. As shown in Theorem 1, a larger h leads to a higher $\prod s_j(v)$, which in turn leads to a higher variance. For example, Folder j in Fig. 2 has $\prod s_j(v) = 4 \times 2 \times 3 \times 3 = 72$, leading to an estimation variance of $72 - 1 = 71$ for the number of files on Level 5 (which has a real value of 1).

4.3 FS_Agg

To reduce the estimation variance, we propose high-level crawling and breadth-first descent to address the two above-described problems on estimation variance, high-level leaf folders, and deep-level folders, respectively. Also, we shall discuss how the variance generated by FS_Agg can be estimated in practice, effectively producing a confidence interval for the aggregate query answer.

4.3.1 High-Level Crawling

It is designed to eliminate the negative impact of high-level leaf folders on estimation variance. The main idea of high-level crawling is to access all folders in the highest i levels of the tree—by following all subfolder branches of folders accessed on or above Level-$(i-1)$. Then, the final estimation becomes an aggregate of two components: the precise value over the highest i levels and the estimated value (produced by random descents) over files below Level-i. One can see from the design of high-level crawling that now leaf folders in the first i levels no longer reduce $p(v)$ for folders v below Level-i (and therefore no longer adversely affect the estimation variance). Formally, we have the following theorem which demonstrates the effectiveness of high-level crawling on reducing the estimation variance:

Theorem 2. *If r_0 out of r folders crawled from the first i levels are leaf folders, then the estimation variance produced by a random descent for the number of Level-h files F_h satisfies*

$$\sigma_{\mathrm{HLC}}(h)^2 \le \frac{(r - r_0) \cdot \sigma(h)^2 - r_0 \cdot F_h^2}{r}. \tag{8}$$

Proof. Before high-level crawling is applied, if the random descent process reaches any of the leaf folders on the first i levels, it in effect returns an estimation of 0 for the number of Level-h files. If r_0 out of r crawled folders are leaf folders, the random descent process has a probability of at least r_0/r to return an estimation of 0 for the number of Level-h files. According to (7), we have

$$\sigma_{\mathrm{HLC}}(h)^2 \le \frac{(\sigma(h)^2 + F_h^2) \cdot (r - r_0)}{r} - F_h^2 \tag{9}$$

$$= \frac{(r - r_0) \cdot \sigma(h)^2 - r_0 \cdot F_h^2}{r}. \tag{10}$$

□

According to this theorem, if we apply high-level crawling over the first level in Fig. 2, then the estimation variance for the number of files on Level 3 is at most $(3 \cdot 24 - 1 \cdot 36)/4 = 9$. As we shall see in Section 5.2 that the variance of estimation after removing Folder c (the only leaf folder at the first level) is exactly 9. Thus, the bound in Theorem 2 is tight in this case.

4.3.2 Breadth-First Descent

It is designed to bring two advantages over FS_Agg_Basic: variance reduction and runtime improvement, which we shall explain as follows:

Variance reduction. Breadth-first descent starts from the root of the tree. Then, at any level of the tree, it generates a set of folders to access at the next level by randomly selecting from subfolders of all folders it accesses at the current level. Any random selection process would work—as long as we know the probability for a folder to be selected, we can answer aggregate queries without bias in the same way as the original random descent process. For example, to COUNT the number of all files in the system, an unbiased estimation of the total number of files at Level i is the SUM of $|v_{i-1}|/p(v_{i-1})$ for all Level-$(i-1)$ folders v_{i-1} accessed by the breadth-first implementation, where $|v_{i-1}|$ and $p(v_{i-1})$ are the number of file branches and the probability of selection for v_{i-1}, respectively.

We use the following random selection process in Glance: consider a folder accessed at the current level which has n_0 subfolders. From these n_0 subfolders, we sample without replacement $\min(n_0, \max(p_{\text{sel}} \cdot n_0, s_{\min}))$ ones for access at the next level. Here, $p_{\text{sel}} \in (0, 1]$ (where sel stands for selection) represents the probability of which a subfolder will be selected for sampling, and $s_{\min} \geq 1$ states the minimum number of subfolders that will be sampled. Both p_{sel} and $s_{\min}$ are user-defined parameters, the settings for which we shall further discuss in the experiments section based on characteristics of real-world file systems.

Compared with the original random descent design, this breadth-first random selection process significantly increases the selection probability for a deep folder. Recall that with the original design, while drilling down one level down the tree, the selection probability can decrease rapidly by a factor of the fan-out (i.e., the number of subfolders) of the current folder. With breadth-first descent, on the other hand, the decrease is limited to at most a factor of $1/p_{\text{sel}}$, which can be much smaller than the fan-out when p_{sel} is reasonably high (e.g., $= 0.5$ as we shall suggest in the experiments section). As a result, the estimation generated by a deep folder becomes much smaller. Formally, we have the following theorem:

Theorem 3. *With breadth-first descent, the variance of estimation on the number of h-level files F_h satisfies*

$$\sigma_{\text{BFS}}(h)^2 \leq \left(\sum_{v \in L_{h-1}} \frac{|v|^2}{p_{\text{sel}}^{h-1}} \right) - F_h^2. \tag{11}$$

Proof. Recall from (7) that

$$\sigma(h)^2 = \left(\sum_{v \in L_{h-1}} \frac{|v|^2}{p(v)} \right) - F_h^2. \tag{12}$$

With breadth-first descent, the probability for the random descent process to reach a node v at Level-h is at least p_{sel}^{h-1}. Thus,

$$\sigma_{\text{BFS}}(h)^2 \leq \left(\sum_{v \in L_{h-1}} \frac{|v|^2}{p_{\text{sel}}^{h-1}} \right) - F_h^2. \tag{13}$$

□

One can see from a comparison with Theorem 1 that the factor of $\prod s_j(v)$ in the original variance, which can grow to an extremely large value, is now replaced by $1/p_{\text{sel}}^{h-1}$ which can be better controlled by the Glance system to remain at a low level even when h is large.

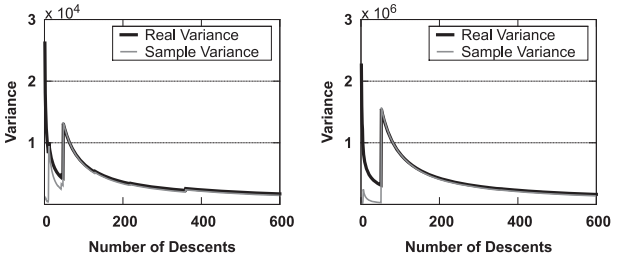
Runtime improvement. In the original design of FS_Agg_Basic, random descent has to be performed multiple times to reduce the estimation variance. Such multiple descents are very likely to access the same folders, especially the high-level ones. While one can leverage the history of hard-drive accesses by caching all historic accesses in memory, such repeated accesses can still take significant CPU time for in-memory lookup. The breadth-first design, on the other hand, ensures that each folder is accessed at most once, reducing the runtime overhead of the Glance system.

4.3.3 Variance Produced by FS_Agg

An important issue for applying FS_Agg in practice is how one can estimate the error of approximate query answers it produces. Since FS_Agg generates unbiased answers for SUM and COUNT queries, the key factor for error estimation here is an accurate computation of the variance. One can see from Theorem 3 that variance depends on the specific structure of the file system, in particular the distribution of selection probability p_{sel} for different folders. Since our sampling-based algorithm does not have a global view of the hierarchical structure, it cannot precisely compute the variance.

Fortunately, the variance can still be accurately *approximated* in practice. To understand how, consider first the depth-first descents used in FS_Agg_Basic. Each descent returns an independent aggregate estimation, while the average for multiple descents becomes the final approximate query answer. Let $\tilde{q}_1, \ldots, \tilde{q}_h$ be the independent estimations and $\tilde{q} = (\sum \tilde{q}_i)/h$ be the final answer. A simple method of variance approximation is to compute $var(\tilde{q}_1, \ldots, \tilde{q}_h)/h$, where $var(\cdot)$ is the variance of independent estimations returned by the descents. Note that if we consider a population consisting of estimations generated by all possible descents, then $\tilde{q}_1, \ldots, \tilde{q}_h$ form a sample of the population. As such, the variance computation is approximating the population variance by sample variance, which are asymptotically equal (for an increasing number of descents).

We conducted extensive experiments described in Section 6 to verify the accuracy of such an approximation. Fig. 3 shows two examples for counting the total number of files in an NTFS and a Plan 9 file system, respectively. Observe from the figure that the real variance oscillates in the beginning of descents. For example, we observe at least one spike on each file system within the first 100 descents. Such a spike occurs when one descent happens to end with a deep-level file which returns an extremely large estimation, and is very likely to happen with our sampling-based technique. Nonetheless, note that the real variance converges to a small value when the number of descents is sufficiently large (e.g., >400). Also note that for two file systems after a small number of descents (about 50), the sample variance $var(\tilde{q}_1, \ldots, \tilde{q}_h)/h$ becomes an extremely accurate approximation for the real (population) variance (overlapping shown in Fig. 3), even during the spikes. One



(a) NTFS (b) Plan 9

Fig. 3. Variance approximation for two file systems, (a) NTFS and (b) Plan 9. Real and sample variances are overlapped when the number of descents is sufficiently large.

can thereby derive an accurate confidence interval for the query answer produced by FS_Agg_Basic.

While FS_Agg no longer performs individual depth-first descents, the idea of using sample variance to approximate population variance still applies. In particular, note that for any given level, say Level-i, of the tree structure, each folder randomly chosen by FS_Agg at Level-$(i-1)$ produces an independent, unbiased, estimation for SUM or COUNT aggregate over all files in Level-i. Thus, the variance for an aggregate query answer over Level-i can be approximated based on the variance of estimations generated by the individual folders. The variance of final SUM or COUNT query answer (over the entire file system) can be approximated by the SUM of variances for all levels.

5 Top-k Query Processing

Recall that for a given file system, a top-k query is defined by two elements: the *scoring function* and the *selection conditions*. Without loss of generality, we consider a top-k query which selects k files (directories) with the *highest* scores. Here, we focus on top-k queries without selection conditions, and consider a tree-like structure of the file system. The extensions to top-k queries with selection conditions and file systems with DAG structures follow in analogy from the same extensions for FS_Agg.

5.1 Main Idea

A simple way to answer a top-k query is to access every directory to find the k files with the highest scores. The objective of FS_TopK is to generate an approximate top-k list with far fewer hard-drive accesses. To do so, FS_TopK consists of the following three steps. We shall describe the details of these steps in the next section.

1. *A lower bound estimation.* The first step uses a random descent similar to FS_Agg to generate an approximate lower bound on the kth highest score over the entire file system (i.e., among files that satisfy the selection conditions specified in the query).
2. *Highest score estimations and tree pruning.* In the second step, we prune the tree structure of the file system according to the lower bound generated in Step 1. In particular, for each subtree, we use the results of descents to generate an upper bound estimate on the highest score of all files in the subtree. If the estimation is smaller than the lower bound from Step 1, we remove the subtree from search space because it is unlikely to contain a top-k file. Note that in order for such a pruning process to have a low false negative rate—i.e., not to falsely remove a large number of real top-k files, a key assumption we are making here is the "locality" of scores—i.e., files with similar scores are likely to colocate in the same directory or close by[3] in the tree structure. Intuitively, the files in a directory are likely to have similar creation and update times. In some cases (e.g., images in the "My Pictures" directory, and outputs from a simulation program), the files will likely have similar sizes too. Note that the strength of this locality is heavily dependent on the type of the query and the semantics of the file system on which the query is running. We plan to investigate this issue as part of the future work.
3. *Crawling of the selected tree.* Finally, we crawl the remaining search space—i.e., the selected tree—by accessing every folder in it to locate the top-k files as the query answer. Such an answer is approximate because some real top-k files might exist in the nonselected subtrees, albeit with a small probability, as we shall show in the experimental results.

In the running example, consider a query for the top-3 files with the highest numbers shown in Fig. 2. Suppose that Step 1 generates a (conservative) lower bound of 8, and the highest scores estimated in Step 2 for subtrees with roots a, c, d, and m are 5, -1 (i.e., no file), 7, and 15, respectively—the details of these estimations will be discussed shortly. Then, the pruning step will remove the subtrees with roots a, c, and d, because their estimated highest scores are lower than the lower bound of 8. Thus, the final crawling step only needs to access the subtree with root of m. In this example, the algorithm would return the files identified as 8, 9, and 10, locating two top-3 files while crawling only a small fraction of the tree. Note that the file with the highest number 11 could not be located because the pruning step removes the subtree with root of d.

5.2 Detailed Design

The design of FS_TopK is built upon a hypothesis that the highest scores estimated in Step 2, when compared with the lower bound estimated in Step 1, can prune a large portion of the tree, significantly reducing the overhead of crawling in Step 3. In the following, we first describe the estimations of the lower bound and the highest scores in Steps 1 and 2, and then discuss the validity of the hypothesis for various types of scoring functions.

Both estimations in the two steps can be made from the *order statistics* [20] of files retrieved by the random descent process in FS_Agg. The reason is that both estimations are essentially on the order statistics of the population (i.e., all files in the system)—the lower bound in Step 1 is the kth largest order statistics of all files, while the highest scores are on the largest order statistics of the subtrees. We refer readers to [20] for details of how the order statistics of a sample can be used to estimate that of the population and how accurate such an estimation is.

3. We define the "distance" between two files as the length of the shortest path connecting the two files in the tree structure. The shorter the distance is, the closer the two files would be.

While sampling for order statistics is a problem in its own right, for the purpose of this paper, we consider the following simple approach which, according to our experiments over real-world file systems, suffices for answering top-k queries accurately and efficiently over almost all tested systems: for the lower bound estimation in Step 1, we use the sample quantile as an estimation of the population quantile. For example, to estimate the 100th largest score of a system with 10,000 files, we use the largest score of a 100-file sample as an estimation. Our tests show that for many practical scoring functions (which usually have a positive skew, as we shall discuss below), the result serves as a conservative lower bound desired by FS_TopK. For the highest score estimation in Step 2, we simply compute $\gamma \cdot \max(\text{sample scores})$, where γ is a constant correction parameter. The setting of γ captures a trade-off between the crawling cost and the chances of finding top-k files—when a larger γ is selected, fewer subtrees are likely to be removed.

We now discuss when the hypothesis of heavy pruning is valid and when it is not. Ideally, two conditions should be satisfied for the hypothesis to hold: 1) If a subtree includes a top-k file, then it should include a (relatively) large number of highly scored files, in order for the sampling process (in Step 2) to capture one (and to thereby produce a highest score estimation that surpasses the lower bound) with a small query cost. And 2) on the other hand, most subtrees (which do not include a top-k file) should have a maximum score significantly lower than the kth highest score. This way, a large number of subtrees can be pruned to improve the efficiency of top-k query processing. In general, one can easily construct a scoring function that satisfies both or neither of the above two conditions. We focus on a special class of scoring functions: those following a heavy-tailed distributions (i.e., its cumulative distribution function $F(\cdot)$ satisfies $\lim_{x\to\infty} e^{\lambda x}(1 - F(x)) = \infty$ for all $\lambda > 0$). Existing studies on real-world file system traces showed that many file/directory metadata attributes, which are commonly used as scoring functions, belong to this category [2]. For example, the distributions of file size, last modified time, creation time, etc., in the entire file system or in a particular subtree are likely to have a heavy tail on one or both extremes of the distribution.

A key intuition is that scoring functions defined as such attribute values (e.g., finding the top-k files with the maximum sizes or the latest modified time) usually satisfy both conditions: first, because of the long tail, a subtree which includes a top-k scored file is likely to include many other highly scored files too. Second, since the vast majority of subtrees have their maximum scores significantly smaller than the top-k lower bound, the pruning process is likely to be effective with such a scoring function.

We would also like to point out an "opposite" class of scoring functions for which the pruning process is not effective: the inverse of the above scoring functions—e.g., the top-k files with the smallest sizes. Such a scoring function, when used in a top-k query, selects k files from the "crowded" light-tailed side of the distribution. The pruning is less likely to be effective because many other folders may have files with similar scores, violating the second condition stated above. Fortunately, asking for top-k smallest files is not particularly useful in practice, also because of the fact that it selects from the crowded side—the answer is likely to be a large number of empty files.

6 Implementation and Evaluation

6.1 Implementation

We implemented Glance, including all three algorithms (FS_Agg_Basic, FS_Agg, and FS_TopK) in 1,600 lines of C code in Linux. We also built and used a simulator in Matlab to complete a large number of tests within a short period of time. While the implementation was built upon the ext3 file system, the algorithms are generic to any hierarchical file system and the current implementation can be easily ported to other platforms, e.g., Windows and Mac OS. FS_Agg_ Basic has only one parameter: the number of descents. FS_Agg has three parameters: the selection probability p_{sel}, the minimum number of selections $s_{\min}$, and the number of (highest) levels for crawling h. Our default parameter settings are $p_{\text{sel}} = 50\,\text{percent}$, $s_{\min} = 3$, and $h = 4$. We also tested with other combinations of parameter settings. FS_TopK has one additional parameter, the (estimation) enlargement ratio γ. The setting of γ depends on the query to be answered, which shall be explained later.

6.2 Experiment Setup

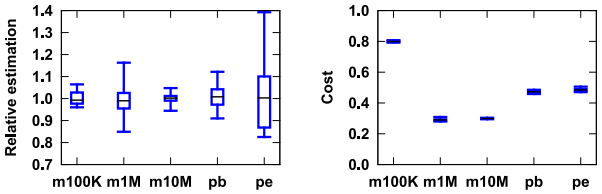
6.2.1 Test Platform

We ran all experiments on Linux machines[4] with Intel Core 2 Duo processor, 4 GB RAM, and 1 TB Samsung 7200RPM hard drive. Unless otherwise specified, we ran each experiment for five times and reported the averages.

6.2.2 Windows File Systems

The Microsoft traces [2] include the snapshots of around 63,000 file systems, 80 percent of which are NTFS and the rest are FAT. For each file system, the trace includes a separate entry for each file and directory in the system, recording its metadata attributes (e.g., size and time stamps). This enabled us to reconstruct the file system by first creating the directory tree structure, and then populating all files into the corresponding directories. We also set the metadata attributes for all files and directories according to the trace entries. To test Glance over file systems with a wide range of sizes, we first selected from the traces two file systems, $m100K$ and $m1M$ (the first "m" stands for Microsoft trace), which are the largest file systems with less than 100K and 1M files, respectively. Specifically, $m100K$ has 99,985 files and 16,013 directories, and $m1M$ has 998,472 files and 106,892 directories. We also tested the largest system in the trace, $m10M$, which has the maximum number of files (9,496,510) and directories (789,097). We put together the largest 33 file systems in the trace to obtain $m100M$ that contains over 100M files and 7M directories. In order to evaluate next-generation billion-level file systems

4. We note that the file-system traces we used are captured from various operating systems—as such, the subtle differences between OS implementations may lead to different performance figures in terms of file-system access time. Nonetheless, the vast majority of our experiments are not directly testing the file-system performance, but the accuracy and access cost measures which are not affected by the specific file system as long as the tree structure is determined.



(a) Accuracy (b) Cost

Fig. 4. Box plots of accuracy and cost of 100 trials.

for which there are no available traces, we chose to replicate $m100M$ for 10 times to create $m1B$ with over one billion files and 70M directories. While a similar scale-up approach has been used in the literature [26], [51], we would like to note that the duplication-filled system may exhibit different properties from a real system with 100M or 1B files. As part of future work, we shall evaluate our techniques in real-world billion-level file systems. Note that we also used other file systems in the trace for testing Glance in a distributed environment.

6.2.3 Plan 9 File Systems

Plan 9 is a distributed file system developed and used at the Bell Labs [43], [44]. We replayed the trace data collected on two central file servers *bootes* and *emelie*, to obtain two file systems, *pb* (for *bootes*) and *pe* (for *emelie*), each of which has over 2M files and 70-80K directories.

6.2.4 NFS

Here, we used the Harvard trace [21], [47] that consists of workloads on NFS servers. The replay of one-day trace created about 1,500 directories and 20K files. Again, we scaled up the one-day system to a larger file system *nfs* (2.3M files and 137K folders), using the above-mentioned approach.

6.2.5 Synthetic File Systems

To conduct a more comprehensive set of experiments on file systems with different file and directory counts, we used *Impressions* [1] to generate a set of synthetic file systems. Impressions takes as input the distributions of file and directory counts and metadata attributes (e.g., number of files per directory, file size, and time stamps), and randomly generates a file system image on disk. For metadata attributes, Impressions by default uses the empirical distributions identified by studies on the Microsoft traces [2]. By adjusting the file count and the (expected) number of files per directory, we used Impressions to generate three file systems, $i10K$, $i100K$, and $i1M$ (here "i" stands for Impressions), with file counts 10K, 100K, and 1M, and directory counts 1K, 10K, and 100K, respectively.

6.3 Aggregate Queries

We first considered Q1 discussed in Section 2, i.e., the total number of files in the system. To provide a more intuitive understanding of query accuracy (than the arguably abstract measure of relative error), we used the Matlab simulator (for quick simulation) to generate a box plot (Fig. 4) of estimations and overhead produced by Glance on Q1 over five file systems, m100K to m10M, pb, and pe. In the figure, the central line of each box represents the median value, and the edges of the box stand for the 25th and 75th percentiles of the runs. Remember as defined in Section 2, the query cost (in Fig. 4b and the following figures) is the ratio between the number of directories visited by Glance and that by file-system crawling. One can see that Glance consistently generates accurate query answers, e.g., for m10M, sampling 30 percent of directories produces an answer with 2 percent average error. While there are outliers, the number of outliers is small and their errors never exceed 7 percent.

We also evaluated Glance with other file systems and varied the input parameter settings. This test was conducted on the Linux and ext3 implementation, and so were the following tests on aggregate queries. In this test, we varied the minimum number of selections $s_{\min}$ from 3 to 6, the number of crawled levels h from 3 to 5, and set the selection probability as $p_{\text{sel}} = 50$ percent (i.e., half of the subfolders will be selected if the amount is more than $s_{\min}$). Fig. 5 shows the query accuracy and cost on the 11 file systems we tested. For all file systems, Glance was able to produce very accurate answers (with <10 percent relative error) when crawling four or more levels (i.e., $h \geq 4$). Also note from Fig. 5 that the performance of Glance is less dependent on the type of the file system than its size—it achieves over 90 percent accuracy for NFS, Plan 9, and NTFS (m10M to m1B). Depending on the individual file systems, the cost ranges from less than 12 percent of crawling for large systems

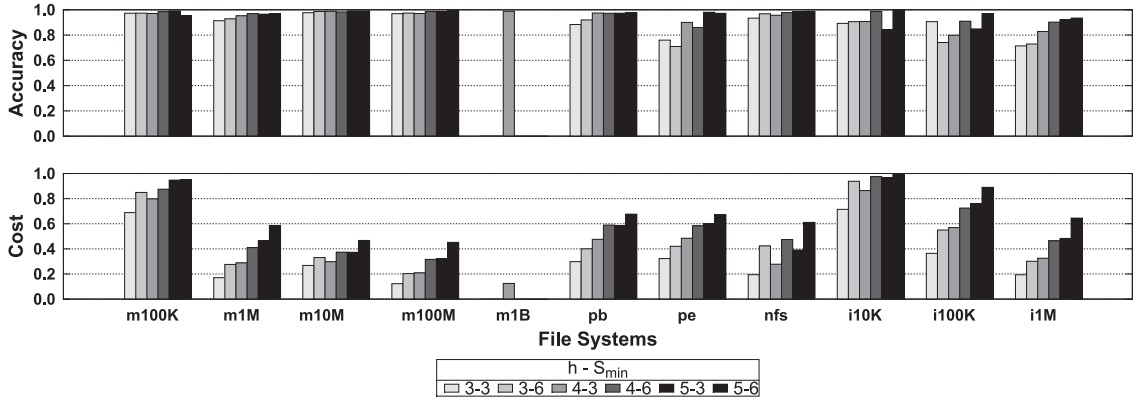
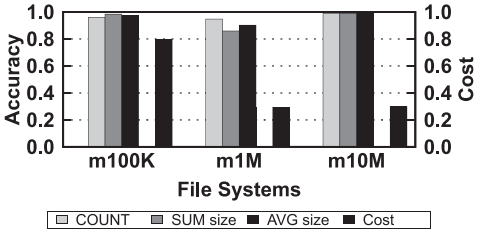
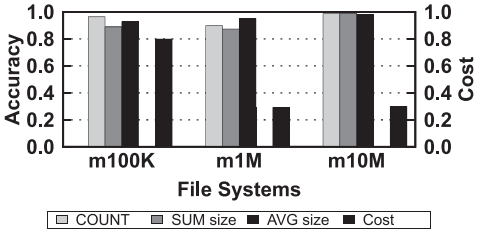


Fig. 5. Accuracy and cost of aggregate queries under different settings of the input parameters. Label 3-3 stands for h of 3 and $s_{\min}$ of 3, 3-6 for h of 3 and $s_{\min}$ of 6, etc., while p_{sel} is 50 percent for all cases.



(a) No condition



(b) With condition

Fig. 6. Accuracy and cost of queries.

with 1B files and 80 percent for the small 100K system. The algorithm scales very well to large file systems, e.g., m100M and m1B—the relative error is only 1-3 percent when Glance accesses only 10-20 percent of all directories. For m1B, the combination of $p_{\text{sel}} = 50$ percent, $s_{\min} = 3$, and $h = 4$ produces 99 percent accuracy with very little cost (12 percent).

The absolute runtime depends heavily on the size of the file system, e.g., seconds for m100K, several minutes for nfs (2.3M files), and 1.2 hours for m100M. Note that in this paper, we only used a single hard drive; parallel IO to multiple hard drives (e.g., RAID) will be able to utilize the aggregate bandwidth to further improve the performance. Due to space limitations, we could not include a detailed discussion of the results. Please refer to [27] for a complete presentation.

We also considered other aggregate queries with various aggregate functions and with/without selection conditions, that is, Q2 and Q3 like queries as in Section 2. Fig. 6a presents the accuracy and cost of evaluating the SUM and AVG of file sizes for all files in the system, while Fig. 6b depicts the same for *exe* files. We included in both figures the accuracy of COUNT because AVG is calculated as SUM/COUNT. Both SUM and AVG queries receive very accurate answers, e.g., only 2 percent relative error for m10M with or without the selection condition of "*.exe*." The query costs are moderate for large systems—30 percent for m1M and m10M (higher for the small system m100K). We also tested SUM and AVG queries with other selection conditions (e.g., file type = "*.dll*") and found similar results.

6.4 Improvements over FS_Agg_Basic

To investigate the effectiveness of the two enhancements used in FS_Agg, we compared the accuracy and efficiency (in terms of runtime) of FS_Agg with FS_Agg_Basic over Q1. Fig. 7 depicts the result, with runtime normalized to that of the *find* command in Linux. One can see from the result that while both algorithms are much more efficient than *find* for almost all file systems, the improvement from FS_Agg_Basic to FS_Agg is also significant—e.g., for

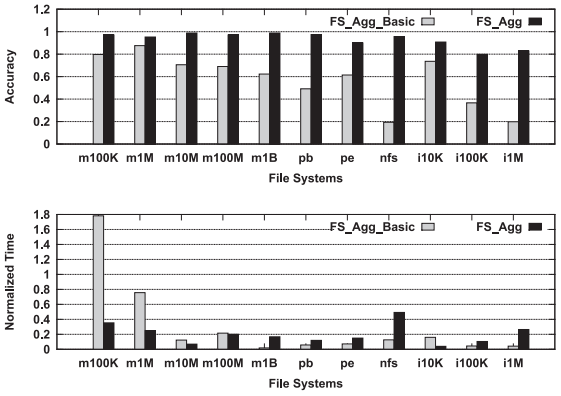
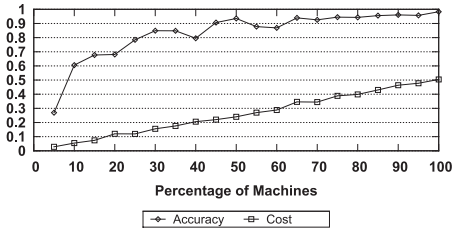


Fig. 7. Query accuracy and time for Basic and FS_Agg.

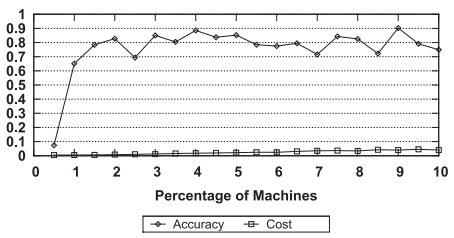
$m100M$, the accuracy increases from 69 to 97 percent while the runtime decreases slightly. Note that while the runtime for FS_Agg is actually higher for $i100K$ and $i1M$, this can be justified by more than 40 percent gains in terms of accuracy.

6.5 Distributed Aggregate Queries

To emulate a distributed environment, we tested FS_Agg over a combination of 100 or 1,000 file systems randomly selected from the Microsoft traces. Note that, to process aggregate queries over a distributed system, Glance may only sample a small percentage of randomly selected machines to perform FS_Agg. To verify the effectiveness of this approach, we varied the *selection percentage*—i.e., the percentage of machines randomly selected to run FS_Agg—from 0 to 100 percent for the 100-machine system and from 0 to 10 percent for the 1,000-machine system. After running FS_Agg over all selected systems, we multiplied the average file count per machine with the total number of machines to produce the final query answer. Fig. 8 depicts the accuracy and cost for counting the total number of files over both systems. For the 100-machine system, the query accuracy increases quickly to above 80 percent when



(a) 100 machines



(b) 1,000 machines

Fig. 8. Distributed queries.

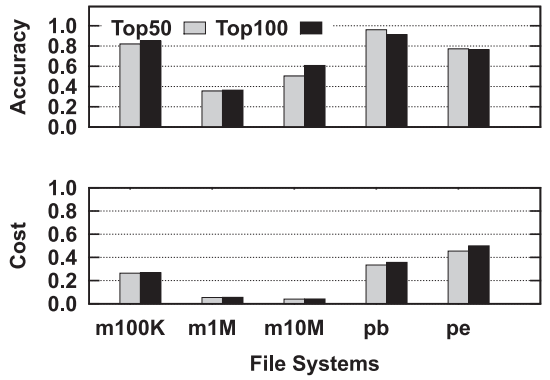


Fig. 9. Accuracy and cost of Top-k queries on file size.

sampling 30 percent of the machines and incurring a cost of 16 percent compared with crawling all machines. The accuracy is further improved to 98 percent when all machines are selected and a query cost of 50 percent is incurred. The performance is even better for the 1,000-machine system. In particular, 80 percent accuracy is achieved when 2 percent of all machines are selected—incurring a query cost of just 0.9 percent. The accuracy varies when sampling more machines, which is not surprising considering the large variations among all file systems.

6.6 Top-k Queries

To evaluate the performance of FS_TopK, we considered both Q5 and Q6 discussed in Section 2. For Q5, i.e., the k largest files, we tested Glance over five file systems, with k being 50 or 100. One can see from the results depicted in Fig. 9 that, in all but one case (m1M), Glance is capable of locating at least 50 percent of all top-k files (for pb, more than 95 percent are located). Meanwhile, the cost is as little as 4 percent of crawling (for m10M). For these top-k queries, similar to aggregate queries, the runtime is correlated with the size of the file system—the queries take only a few seconds for small file systems, and up to 10 minutes for large systems (e.g., m10M). When we varied γ from 1, 5, 10 to 100,000 in this test, we found that the query cost increases as γ does. Fortunately, a moderate γ of 5 and 10 presents a good trade-off point—achieving a reasonable accuracy without incurring too much cost.

We also tested Q6, i.e., the k most recently modified files over m100K, m1M, and pb, and Glance is capable of locating more than 90 percent of top-k files for pb, and about 60 percent for m100K and m1M. The cost, meanwhile, is 28 percent of crawling for m100K, 1 percent for m1M, and 36 percent for pb. Interested readers may refer to [27] for details.

7 Related Work

7.1 Metadata Query on File Systems

Prior research on file-system metadata query [26], [28], [34], [36] has extensively focused on databases, which utilize indexes on file metadata. However, the results [26], [33], [34] reviewed the inefficiency of this paradigm due to metadata locality and distribution skewness in large file systems. To solve this problem, Spyglass [32], [34], SmartStore [26], and Magellan [33] utilize multidimensional structures (e.g., K-D trees and R-trees) to build indexes upon subtree partitions or semantic groups. SmartStore attempts to reorganize the files based on their metadata semantics. Conversely, Glance avoids any file-specific optimizations, aiming instead to maintain file system agnosticism. It works seamlessly with the tree structure of a file system and avoids the time and space overheads from building and maintaining the metadata indexes.

7.2 Comparison with Database Sampling

Traditionally, database sampling has been used to reduce the cost of retrieving data from a DBMS. Random sampling mechanisms have been extensively studied [4], [6], [9], [12], [14], [15], [22], [37]. Applications of random sampling include estimation methodologies for histograms and approximate query processing (see tutorial in [15]). However, these techniques do not apply when there is no direct random access to all elements of interest—e.g., in a file system, where there is no complete list of all files/directories.

Another particularly relevant topic is the sampling of hidden web databases [8], [24], [25], [30], for which a random descent process has been used to construct queries issued over the web interfaces of these databases [16], [17], [18], [19]. While both these techniques and Glance use random descents, a unique challenge for sampling a file system is its much more complex distribution of files. If we consider a hidden database in the context of a file system, then all files (i.e., tuples) appear under folders with no subfolders. Thus, the complex distribution of files in a file system calls for a different sampling technique which we present in the paper.

7.3 Top-k Query Processing

Top-k query processing has been extensively studied over both databases (e.g., see a recent survey [29]) and file systems [3], [7], [26], [34]. For file systems, a popular application is to locate the top-k most frequent (or space-consuming) files/blocks for redundancy detection and removal. For example, Lillibridge et al. [35] proposed the construction of an in-memory sparse index to compare an incoming block against a few (most similar) previously stored blocks for duplicate detections (which can be understood as a top-k query with a scoring function of similarity). Top-k query processing has also been discussed in other index building techniques, e.g., in Spyglass [34] and SmartStore [26].

8 Discussion

At present, Glance takes several predefined parameters as the inputs and needs to complete the execution in whole. That is, Glance is not an anytime algorithm and cannot be stopped in the middle of the execution, because our current approach relies on a complete sample to reduce query variance and achieve high accuracy. One limitation of this approach is that its runtime over an alien file system is unknown in advance, making it unsuitable for the applications with absolute time constraints. For example, a border patrol agent may need to count the amount of encrypted files in a traveler's hard drive, in order to determine whether the traveler could be transporting sensitive documents across the border [13], [46]. In this case, the agent

must make a fast decision as the amount of time each traveler can be detained for is extremely limited. We envision that in the future, Glance shall offer a time-out knob that a user can use to decide the query time over a file system. This calls for new algorithms that allow Glance to get smarter—be predictive about the runtime and self-adjust the work flow based on the real-time requirements.

Glance currently employs a "static" strategy over file systems and queries, i.e., it does not modify its techniques and traversals for a query. A dynamic approach is attractive because in that case Glance would be able to adjust the algorithms and parameters depending on the current query and file system. New sampling techniques, e.g., stratified and weighted sampling, shall be investigated to further improve query accuracy on large file systems. The semantic knowledge of a file system can also help in this approach. For example, most images can be found in a special directory, e.g., "/User/Pictures/" in MacOS X, or "\Documents and Settings\User\My Documents\My Pictures\" in Windows XP.

Glance shall also leverage the results from the previous queries to significantly expedite the future ones, which is beneficial in situations when the workload is a set of queries that are executed very infrequently. The basic idea is to store the previous estimations over parts (e.g., subtrees) of the file system, and utilize the history to limit the search space to the previously unexplored part of the file system, unless it determines that the history is obsolete (e.g., according to a predefined validity period). Note that the history shall be continuously updated to include newly discovered directories and to update the existing estimations.

9 Conclusion

In this paper, we have initiated an investigation of just-in-time analytics over a large-scale file system through its tree- or DAG-like structure. We proposed a random descent technique to produce unbiased estimations for SUM and COUNT queries and accurate estimations for other aggregate queries, and a pruning-based technique for the approximate processing of top-k queries. We proposed two improvements, high-level crawling and breadth-first descent, and described a comprehensive set of experiments which demonstrate the effectiveness of our approach over real-world file systems.

Acknowledgments

The authors thank the anonymous reviewers from *IEEE Transactions on Computers*, FAST '11 reviewers, and our FAST shepherd John Bent, for their suggestions that helped significantly improve this paper. They owe them a great deal of gratitude. They also thank Hong Jiang and Yifeng Zhu for their help on replaying the NFS trace, and Ron C. Chiang for his help on the artwork. This work was supported by the US National Science Foundation (NSF) grants OCI-0937875, OCI-0937947, IIS-0845644, CCF-0852674, CNS-0852673, and CNS-0915834. A preliminary version of this paper appeared in the 9th USENIX Conference on File and Storage Technologies (FAST '11).

References

[1] N. Agrawal, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Generating Realistic Impressions for File-System Benchmarking," *ACM Trans. Storage,* vol. 5, no. 4, pp. 1-30, 2009.

[2] N. Agrawal, W. Bolosky, J. Douceur, and J. Lorch, "A Five-Year Study of File-System Metadata," *Proc. Fifth USENIX Conf. File and Storage Technologies,* pp. 31-45, 2007.

[3] S. Ames, M. Gokhale, and C. Maltzahn, "Design and Implementation of a Metadata-Rich File System," Technical Report UCSC-SOE-10-07, Univ. of California, Santa Cruz, 2010.

[4] D. Barbara, "The New Jersey Data Reduction Report," *IEEE Data Eng. Bull.,* vol. 20, no. 4, pp. 3-45, Dec. 1997.

[5] Beagle, http://beagle-project.org/, 2011.

[6] J. Bethel, "Sample Allocation in Multivariate Surveys," *Survey Methodology,* vol. 15, no. 1, pp. 47-57, 1989.

[7] S. Brandt, C. Maltzahn, N. Polyzotis, and W.-C. Tan, "Fusing Data Management Services with File Systems," *Proc. Fourth Ann. Workshop Petascale Data Storage (PDSW '09),* pp. 42-46, 2009.

[8] J. Callan and M. Connell, "Query-Based Sampling of Text Databases," *ACM Trans. Information Systems,* vol. 19, pp. 97-130, Apr. 2001.

[9] B. Causey, "Computational Aspects of Optimal Allocation in Multivariate Stratified Sampling," *SIAM J. Scientific and Statistical Computing,* vol. 4, pp. 322-329, 1983.

[10] S. Chaudhuri, G. Das, and V. Narasayya, "Optimized Stratified Sampling for Approximate Query Processing," *ACM Trans. Database Systems,* vol. 32, no. 2, p. 9, 2007.

[11] S. Chaudhuri, G. Das, and U. Srivastava, "Effective Use of Block-Level Sampling in Statistics Estimation," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 287-298, 2004.

[12] J. Chromy, "Design Optimization with Multiple Objectives," *Proc. Research Methods of the Am. Statistical Assoc.,* pp. 194-199, 1987.

[13] CNet, "Security Guide to Customs-Proofing Your Laptop," http://news.cnet.com/8301-13578_3-9892897-38.html, 2009.

[14] W. Cochran, *Sampling Techniques.* John Wiley & Sons, 1977.

[15] G. Das, "Survey of Approximate Query Processing Techniques (Tutorial)," *Proc. Int'l Conf. Scientific and Statistical Database Management (SSDBM '03),* 2003.

[16] A. Dasgupta, G. Das, and H. Mannila, "A Random Walk Approach to Sampling Hidden Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '07),* pp. 629-640, 2007.

[17] A. Dasgupta, X. Jin, B. Jewell, N. Zhang, and G. Das, "Unbiased Estimation of Size and Other Aggregates over Hidden Web Databases," *Proc. Int'l Conf. Management of Data (SIGMOD),* pp. 855-866, 2010.

[18] A. Dasgupta, N. Zhang, and G. Das, "Leveraging Count Information in Sampling Hidden Databases," *Proc. IEEE Int'l Conf. Data Eng.,* pp. 329-340, 2009.

[19] A. Dasgupta, N. Zhang, G. Das, and S. Chaudhuri, "Privacy Preservation of Aggregates in Hidden Databases: Why and How?" *Proc. 35th SIGMOD Int'l Conf. Management of Data,* pp. 153-164, 2009.

[20] H.A. David and H.N. Nagaraja, *Order Statistics,* third ed. Wiley, 2003.

[21] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer, "Passive Nfs Tracing of Email and Research Workloads," *Proc. Second USENIX Conf. File and Storage Technologies (FAST '03),* pp. 203-216, 2003.

[22] M.N. Garofalakis and P.B. Gibbon, "Approximate Query Processing: Taming the Terabytes," *Proc. 27th Int'l Conf. Very Large Data Bases (VLDB),* 2001.

[23] Google, Google Desktop, http://desktop.google.com/, 2011.

[24] Y.L. Hedley, M. Younas, A. James, and M. Sanderson, "A Two-Phase Sampling Technique for Information Extraction from Hidden Web Databases," *Proc. Sixth Ann. ACM Int'l Workshop Web Information and Data Management (WIDM '04),* pp. 1-8, 2004.

[25] Y.L. Hedley, M. Younas, A.E. James, and M. Sanderson, "Sampling, Information Extraction and Summarisation of Hidden Web Databases," *Data and Knowledge Eng.,* vol. 59, no. 2, pp. 213-230, 2006.

[26] 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," *Proc. Conf. High Performance Computing Networking, Storage and Analysis (SC),* pp. 1-12, 2009.

[27] H. Huang, N. Zhang, W. Wang, G. Das, and A. Szalay, "Just-in-Time Analytics on Large File Systems," *Proc. Ninth USENIX Conf. File and Storage Technologies,* 2011.

[28] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. Ganger, E. Riedel, and A. Ailamaki, "Diamond: A Storage Architecture for Early Discard in Interactive Search," *Proc. USENIX Conf. File and Storage Technologies (FAST),* 2004.
[29] I.F. Ilyas, G. Beskales, and M.A. Soliman, "A Survey of Top-*k* Query Processing Techniques in Relational Database Systems," *ACM Computing Surveys,* vol. 40, no. 4, pp. 1-58, 2008.
[30] P.G. Ipeirotis and L. Gravano, "Distributed Search over the Hidden Web: Hierarchical Database Sampling and Selection," *Proc. 28th Int'l Conf. Very Large Data Bases (VLDB '02),* pp. 394-405, 2002.
[31] P. Kogge, "Exascale Computing Study: Technology Challenges in Achieving Exascale Systems," *DARPA Information Processing Techniques Office,* vol. 28, 2008.
[32] A. Leung, "Organizing, Indexing, and Searching Large-Scale File Systems," Technical Report UCSC-SSRC-09-09, Univ. of California, Santa Cruz, Dec. 2009.
[33] A. Leung, I. Adams, and E. Miller, "Magellan: A Searchable Metadata Architecture for Large-Scale File Systems," Technical Report UCSC-SSRC-09-07, Univ. of California, Santa Cruz, Nov. 2009.
[34] A.W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E.L. Miller, "Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems," *Proc. Seventh Conf. File and Storage Technologies (FAST),* pp. 153-166, 2009.
[35] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble, "Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality," *Proc. Seventh Conf. File and Storage Technologies (FAST),* pp. 111-123, 2009.
[36] L. Liu, L. Xu, Y. Wu, G. Yang, and G. Ganger, "Smartscan: Efficient Metadata Crawl for Storage Management Metadata Querying in Large File Systems," Carnegie Mellon Univ. Parallel Data Lab Technical Report CMU-PDL-10-112, 2010.
[37] S. Lohr, *Sampling: Design and Analysis.* Cengage Learning, 1999.
[38] N. Murphy, M. Tonkelowitz, and M. Vernal, "The Design and Implementation of the Database File System," http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.11.8068, 2002.
[39] J. Nunez, "High End Computing File System and IO R&D Gaps Roadmap," *Proc. HEC FSIO R&D Conf.,* Aug. 2008.
[40] F. Olken and D. Rotem, "Simple Random Sampling from Relational Databases," *Proc. 12th Int'l Conf. Very Large Data Bases,* pp. 160-169, 1986.
[41] F. Olken and D. Rotem, "Random Sampling from Database Files: A Survey," *Proc. Fifth Int'l Conf. Statistical and Scientific Database Management,* pp. 92-111, 1990.
[42] M. Olson, "The Design and Implementation of the Inversion File System," *Proc. Winter 1993 USENIX Technical Conf.,* pp. 205-217, 1993.
[43] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom, "Plan 9 from Bell Labs," *Computing Systems,* vol. 8, no. 3, pp. 221-254, 1995.
[44] Plan 9 File System Traces, http://pdos.csail.mit.edu/p9trace/, 2011.
[45] M. Seltzer and N. Murphy, "Hierarchical File Systems Are Dead," *Proc. 12th Conf. Hot Topics in Operating Systems (HotOS '09),* p. 1, 2009.
[46] SlashDot, "Laptops Can Be Searched at the Border," http://yro.slashdot.org/article.pl?sid=08/04/22/1733251, 2008.
[47] SNIA, NFS Traces, http://iotta.snia.org/traces/list/NFS, 2010.
[48] P. Stahlberg, G. Miklau, and B.N. Levine, "Threats to Privacy in the Forensic Analysis of Database Systems," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '07)* pp. 91-102, 2007.
[49] A. Szalay, "New Challenges in Petascale Scientific Databases," *Proc. 20th Int'l Conf. Scientific and Statistical Database Management (SSDBM '08),* p. 1, 2008.
[50] J. Vitter, "Random Sampling with a Reservoir," *ACM Trans. Math. Software,* vol. 11, no. 1, pp. 37-57, 1985.
[51] Y. Zhu, H. Jiang, J. Wang, and F. Xian, "HBA: Distributed Metadata Management for Large Cluster-Based Storage Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 19, no. 6, pp. 750-763, June 2008.

H. Howie Huang received the PhD degree in computer science from the University of Virginia. He is an assistant professor in computer engineering in the Department of Electrical and Computer Engineering at the George Washington University. His research interest is computer and systems architecture, especially data-intensive computing, file and storage systems, and grid and cloud computing. His research is supported by awards from US National Science Foundation (NSF) and IBM. He won an IBM Real Time Innovation Faculty Award in 2008.

Nan Zhang received the BS degree from Peking University in 2001 and the PhD degree from Texas A&M University in 2006, both in computer science. He is an assistant professor of computer science at the George Washington University, Washington, District of Columbia. Prior to joining GWU, he was an assistant professor of computer science and engineering at the University of Texas at Arlington from 2006 to 2008. His current research interests span security and privacy issues in databases, data mining, and computer networks, including privacy and anonymity in data collection, publishing, and sharing, privacy-preserving data mining, and wireless network security and privacy. He received the US National Science Foundation (NSF) CAREER Award in 2008.

Wei Wang graduated from Huazhong University of Science and Technology, China, with the BE degree in computer science and received the MS degree in computer science from the University of Delaware (UD). He is working toward the PhD degree at the University of Delaware. Before he continued his PhD program, he spent six months at The George Washington University as a research assistant. His research interests are operating systems, compilers, and high performance computing.

Gautam Das graduated with the BTech degree in computer science from IIT Kanpur, India, and received the PhD degree in computer science from the University of Wisconsin-Madison. He is a full professor in the Computer Science and Engineering Department of the University of Texas at Arlington. Prior to UTA, he has held positions at Microsoft Research, Compaq Corporation, and the University of Memphis, as well as visiting positions at IBM Research. His research interests span data mining, information retrieval, databases, applied graph and network algorithms, and computational geometry. His research has resulted in numerous papers that have appeared in premier conferences and journals in databases, data mining, and theoretical computer science. His research has been supported by grants from federal and state agencies such as the US National Science Foundation (NSF), Office of Naval Research, Department of Education, Texas Higher Education Coordination Board, as well as industry such as Nokia, Microsoft, Cadence, and Apollo.

Alexander S. Szalay is the Alumni Centennial professor of astronomy at the Johns Hopkins University. He is also a professor in the Department of Computer Science. He is a cosmologist, working on the statistical measures of the spatial distribution of galaxies and galaxy formation. He has written papers covering areas from theoretical cosmology to observational astronomy, spatial statistics, and computer science. He is a corresponding member of the Hungarian Academy of Sciences, a fellow of the American Academy of Arts and Sciences. In 2004, he received an Alexander Von Humboldt Award in Physical Sciences, and in 2008, a Microsoft Award for Technical Computing.