Just-In-Time Analytics on Large File Systems

H. Howie Huang[1], Nan Zhang[1], Wei Wang[1], Gautam Das[2], and Alexander S. Szalay[3]

[1]George Washington University
[2]University of Texas at Arlington
[3]Johns Hopkins University

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 pre-processing 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.

1 Introduction

Today a file system with billions of files, millions of directories and petabytes of storage is no longer an exception [29]. As file systems grow, users and administrators are increasingly keen to perform complex queries [37, 47], such as "How many files have been updated since ten days ago?", 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 are often needed by applications that require *just-in-time analytics* over large file systems, such as data management, archiving, enterprise surveillance, etc. The just-in-time requirement is defined by two properties: (1) file-system analytics must be completed within a short amount of time, and (2) the analyzer holds no prior knowledge (e.g., pre-processing 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 [43]. 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) scanning 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% 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

within a short amount of time. (2) utilizing pre-built indexes which are regularly updated [3, 7, 26, 32, 36, 40]. Many desktop search products, e.g., Google Desktop [23] and Beagle [5], belong to this category. 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:

- Index building can be unrealistic for many applications that require just-in-time analytics over an alien file system. An example is enterprise surveillance [35], where portable machines and storage devices must be quickly examined before being allowed to join the enterprise network.
- Even if index can be built up-front, its significant cost may not be justifiable if the index is not frequently used afterwards. Unfortunately, this is common for some large file systems, e.g., storage archives or scratch data for scientific applications rarely 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 [46] 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. In general, 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 pre-processing 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 [39] - 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 [38], stratified [10], reservoir [48] 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).

Our main contributions are two-fold: (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% accuracy at 20% 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 so 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 meta-data attributes in real-world file systems [2].

The rest of the paper is organized as follows. Section 2 presents the problem definition. In Section 3 and 4, we describe FS_Agg and FS_TopK for processing aggregate and top-k queries, respectively. The evaluation results are shown in Section 5. Section 6 reviews the related work, followed by the conclusion in Section 7.

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.

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, timestamp) 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;

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);*

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 ten 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.

Problem Statement (Objective of Just-In-Time Analytics over File Systems): 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. 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 is, more directories the algorithm has to sample, leading to a longer runtime.

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. For example, while scientific computing usually requires a small error, the above-mentioned surveillance application may simply need a ball-park figure to determine whether there is a significant amount of sensitive files in the system.

3 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.

3.1 FS_Agg_Basic

A Random Descent Process: In general, the folder organization of a file system can be considered as a tree or a directed acyclic graph (DAG), 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 subsection.

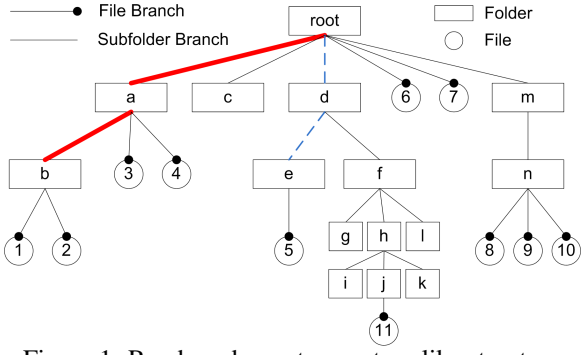


Figure 1: Random descents on a tree-like structure

Figure 1 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 numbers we recorded during such a descent process are $f_0, s_0, f_1, s_1, \ldots, f_h, s_h$, where $s_h = 0$ because each descent ends at a leaf-folder. We estimate the COUNT of all files as

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

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 Figure 1 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).

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 i-th level (with root being Level 0) of the tree (e.g., Files 1 and 2 in Figure 1 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

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

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

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

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.

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.

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 Figure 1. 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.

AVG: A simple way to process an AVG query is to estimate the corresponding SUM and COUNT respectively, and then 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 indeed 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.

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 can terminate immediately because any deeper descent can only return $f_i = 0$, leading to no change in the estimation.

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, NTFS) provide a *reference count* for each file which indicates the number of links in the DAG that point to the file[1]. 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 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.

3.2 Disadvantages of FS_Agg_Basic

While the estimations generated by FS_Agg_Basic is 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*

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

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

[1] 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.

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

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

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

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

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 Figure 1 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 Figure 1 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 Figure 1, 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 Figure 1. The key problem arising from such a folder is that the probability for it to 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 Figure 1 has $\prod s_j(v) = 4 \times 2 \times 3 \times 3 = 72$, leading to a estimation variance of $72 - 1 = 71$ for the number of files on Level 5 (which has a real value of 1).

3.3 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.

High-Level Crawling 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[2] 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*

$$\sigma_{\text{HLC}}(h)^2 \leq \frac{(r - r_0) \cdot \sigma_h^2 - r_0 \cdot F_h^2}{r}. \quad (8)$$

According to this theorem, if we apply high-level crawling over the first level in Figure 1, then the estimation variance for the number of files on Level 3 is at most $(3 \cdot 24 - 1 \cdot 36)/4 = 9$. Recall from Section 4.2 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.

Breadth-First Descent 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 currently level. Note that 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 $\min(n_0, \max(p_{\text{sel}} \cdot$

[2] In the rest part of the paper, we do not include the proof of theorems due to the space limitation.

$n_0, s_{\min}))$ 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 fanout 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*

$$\sigma_{\text{BFS}}(h)^2 \leq \left(\sum_{v \in L_{h-1}} \frac{|v|^2}{p_{\text{sel}}^{h-1}} \right) - F_h^2. \qquad (9)$$

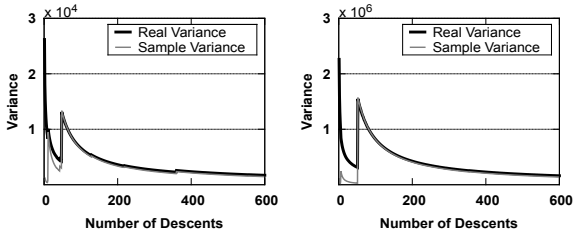
One can see from a comparison with Theorem 1 that the factor of $\prod s_j(v)$ in the original variance, which can grow to an extremely large value, is now replaced by $1/p_{\text{sel}}^{h-1}$ which can be better controlled by the Glance system to remain at a low level even when h is large.

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 look up. 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.

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 enabling 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 5 to verify the accuracy of such an approximation. Figure 2 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 Figure 2), even during the spikes. One can thereby derive an accurate confidence interval for the query answer produced by FS_Agg_Basic.



(a) NTFS (b) Plan 9

Figure 2: Variance approximation for (a) an NTFS file system and (b) a Plan 9 system. Real and sample variances are overlapped when the number of descents is sufficiently large.

While FS_Agg no longer performs individual depth-first descents, the idea of using sample variance to ap-

proximate 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 then be approximated by the SUM of variances for all levels.

4 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. For the sake of simplicity, 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.

4.1 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 subsection.

1. *A Lower-Bound Estimation:* The first step uses a random descent similar to FS_Agg to generate an approximate lower bound on the k-th 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 co-locate in the same directory or close by 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 selected 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 Figure 1. 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 a. 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 here because the pruning step removes the subtree with root of d.

4.2 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 k-th 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.

While sampling for order statistics is a problem of 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 100-th 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 tradeoff between the crawling cost and the chances of finding top-k files - when a larger γ is selected, less number of the subtrees are likely 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 k-th 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 satisfy 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 here 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 as well. 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 - e.g., the answer is likely to be a large number of empty files.

5 Implementation and Evaluation

5.1 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_{sel} = 50%, $s_{\min}$ = 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.

5.2 Experiment Setup

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

Windows File Systems: The Microsoft traces [2] includes the snapshots of around 63,000 file systems, 80% of which are NTFS and the rest are FAT. 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 for which there are no available traces, we chose to replicate $m100M$ for 10 times to create $m1B$ with over 1 billion files and 70M directories. While a similar scale-up approach has been used in the literature [26,49], 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.

Plan 9 File Systems: Plan 9 is a distributed file system developed and used at the Bell Labs [41, 42]. 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.

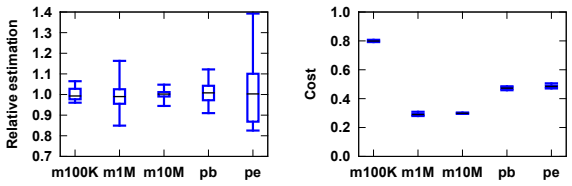
NFS: Here we used the Harvard trace [21, 45] that consists of workloads on NFS servers. The replay of one-day trace created about 1,500 directories and 20K files. Again, we scaled up the one-day system to a larger file system *nfs* (2.3M files and 137K folders), using the above-mentioned approach.

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. 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.

5.3 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 (Figure 3(a)) of estimations and overhead produced by Glance on Q1 over five file systems, m100K to m10M, pb and pe. Remember as defined in Section 2, the query cost (in Figure 3(b) 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% of directories produces an answer with 2% average error. While there are outliers, the number of outliers is small and their errors never exceed 7%.

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_{sel} = 50% (i.e., half of the subfolders will be selected if the amount is more than $s_{\min}$). Figure 4 shows the query accuracy and cost on the eleven file systems we tested. For all file systems, Glance was able to produce very accurate answers (with <10% relative error) when crawling four or more levels (i.e., $h \geq 4$). Also note from Figure 4 that the performance of Glance is less dependent on the type of the file system than its size - it achieves over 90% accuracy for NFS, Plan 9, and NTFS (m10M to m1B). Depending on the individual file systems, the cost ranges from less than 12% of crawling for large systems with 1B files and 80% 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% when Glance accesses only 10-20% of all directories. For m1B, the combination of p_{sel} = 50%, $s_{\min}$ = 3 and h = 4 produces 99% accuracy with very little cost (12%).



(a) Accuracy (b) Cost

Figure 3: Box plots of accuracy and cost of 100 trials

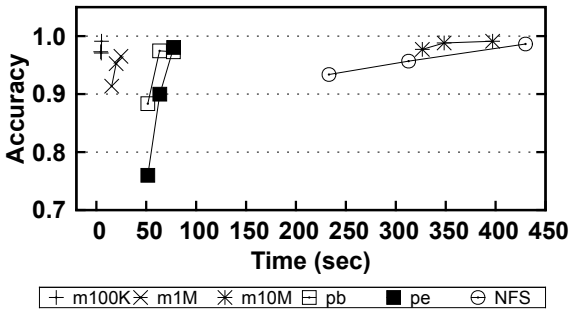


Figure 5: Query accuracy vs. run time in seconds. Three points of each line (from left to right) represent h of 3, 4, and 5, respectively.

Figure 5 illustrates the runtimes (in seconds) for aggregate queries. 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 (not shown in the figure). 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. As the value of h increases, the query runs slightly longer

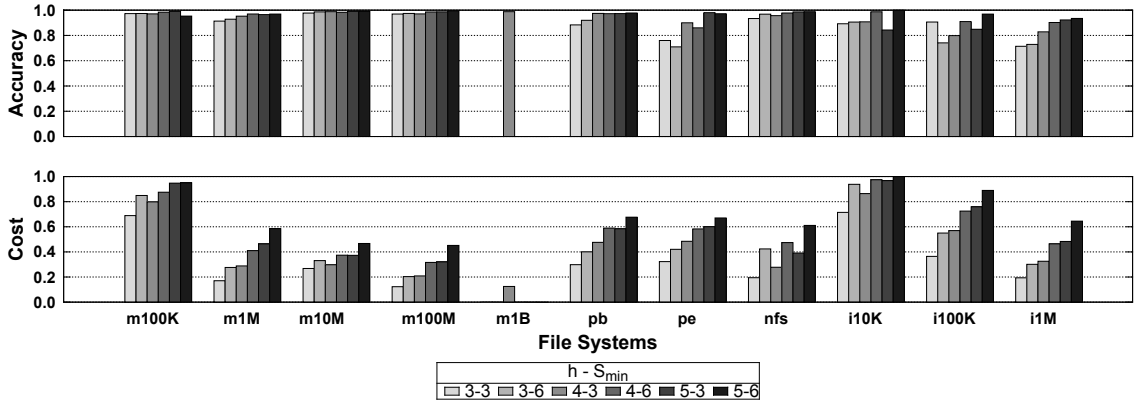
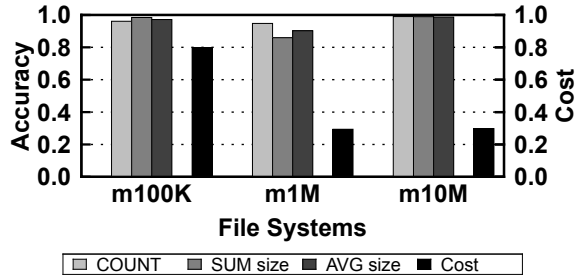
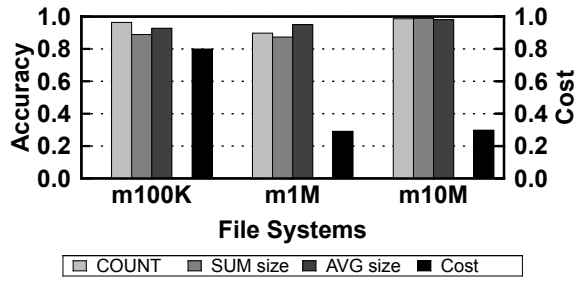


Figure 4: Accuracy and cost of aggregate queries under different settings of the input parameters. Label 3-3 stands for h of 3 and $s_{\min}$ of 3, 3-6 for h of 3 and $s_{\min}$ of 6, etc., while p_{sel} is 50% for all cases.

but the accuracy improves by about 10% for pb and 20% for pe. The accuracy improvements for m10M and nfs are smaller. The value of $s_{\min}$ is 3 in this test.



(a) No condition



(b) With condition

Figure 6: Accuracy and cost of queries

We also considered other aggregate queries with various aggregate functions and with/without selection conditions. Figure 6(a) presents the accuracy and cost of evaluating the SUM and AVG of file sizes for all files in the system, while Figure 6(b) 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% relative error for m10M with or without the selection condition of '*.exe*'. The query costs are moderate for large systems - 30% 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.

5.4 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 Figure 7 that, in all but one case (m1M), Glance is capable of locating at least 50% of all top-k files (for pb, more than 95% are located). Meanwhile, the cost is as little as 4% of crawling (for m10M). Figure 8 presents the runtimes of the top-k queries, where one can see that similar to aggregate queries, the runtime is correlated to the size of the file system - the queries take only a few seconds for small file systems, and up to ten minutes for large systems (e.g., m10M).

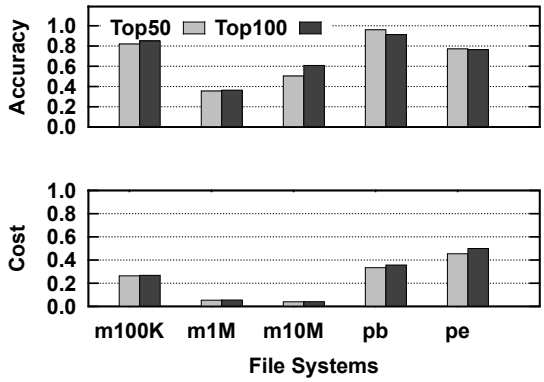


Figure 7: Accuracy and cost of Top-k queries on file size

Figure 9 presents the query accuracy and cost for Top-k queries on file size, when γ varies from 1, 5, 10, to 100,000. The trend is clear - the query cost increases as γ does, because a higher value of γ is to scale the highest-score estimation up to a larger degree, that is, to crawl a

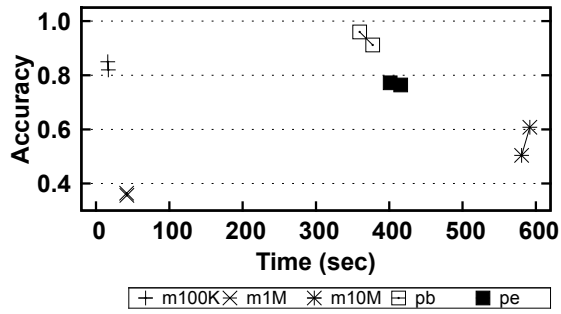


Figure 8: Top-k query accuracy vs. run time in seconds. The first point of each line stands for top-50 and the second for top-100.

larger portion of the file system. Fortunately, a moderate γ of 5 and 10 presents a good tradeoff 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. The results are shown in Figure 10. One can see that Glance is capable of locating more than 90% of top-k files for pb, and about 60% for m100K and m1M. The cost, meanwhile, is 28% of crawling for m100K, 1% for m1M, and 36% for pb.

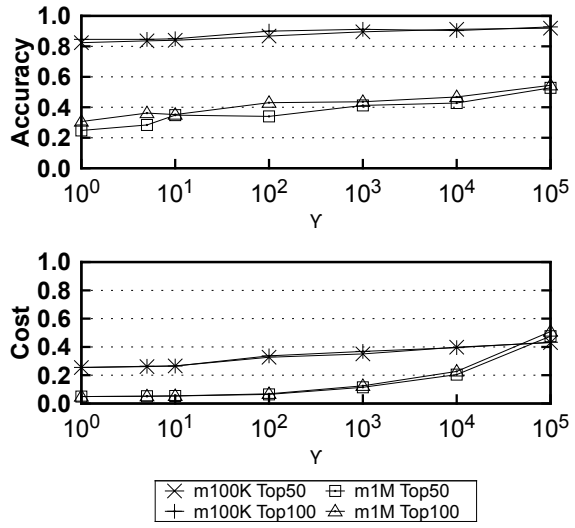


Figure 9: Query accuracy and cost when varying γ

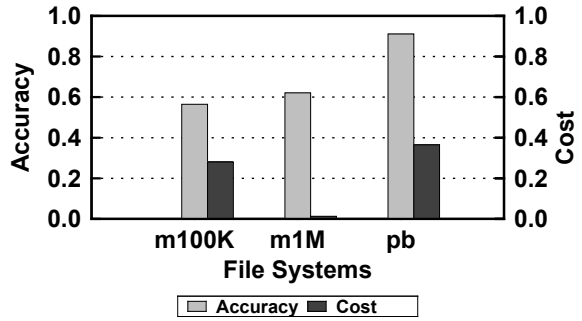


Figure 10: Top-k queries on file time

6 Related Work

Metadata query on file systems: Prior research on file-system metadata query [26, 32] has extensively focused on databases, which utilizes indexes on file metadata. However, the results [26,31,32] reviewed the inefficiency of this paradigm due to metadata locality and distribution skewness in large file systems. To solve this problem, Spyglass [30, 32], SmartStore [26], and Magellan [31] utilize multi-demensional 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.

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, 34]. 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, 28], for which a random descent process has been used to construct queries issued over the web interfaces of these databases [16–19]. 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 .

Top-k Query Processing: Top-k query processing has been extensively studied over both databases (e.g., see a recent survey [27]) and file systems [3, 7, 26, 32]. 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. [33] 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 [32] and SmartStore [26].

7 Discussion

At present, Glance takes several pre-defined parameters as the inputs and needs to complete the execution in whole. That is, Glance is not an any-time 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, 44]. 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 get smarter - be predictive about the run-time 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 pre-defined validity period). Note that the history shall be continuously updated to include newly discovered directories and to update the existing estimations.

8 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.

9 Acknowledgments

We thank the anonymous reviewers and our shepherd John Bent for their excellent comments that helped improve the quality of this paper. We also thank Hong Jiang and Yifeng Zhu for their help on replaying the NFS trace, and Ron Chiang for his help on the artwork. This work was supported by the NSF grants OCI-0937875, OCI-0937947, IIS-0845644, CCF-0852674, CNS-0852673, and CNS-0915834.

References

[1] AGRAWAL, N., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. Generating realistic impressions for file-system benchmarking. *ACM Transactions on Storage (TOS) 5*, 4 (2009), 1–30.

[2] AGRAWAL, N., BOLOSKY, W., DOUCEUR, J., AND LORCH, J. A five-year study of file-system metadata. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies* (2007), pp. 31–45.

[3] AMES, S., GOKHALE, M., AND MALTZAHN, C. Design and implementation of a metadata-rich file system. Tech. Rep. UCSC-SOE-10-07, University of California, Santa Cruz, 2010.

[4] BARBARA, D., DUMOUCHEL, W., FALOUTSOS, C., HAAS, P., HELLERSTEIN, J., IOANNIDIS, Y., JAGADISH, H., JOHNSON, T., NG, R., POOSALA, V., ET AL. The New Jersey data reduction report. *IEEE Data Eng. Bull. 20*, 4 (1997), 3–45.

[5] BEAGLE. *http://beagle-project.org/*.

[6] BETHEL, J. Sample allocation in multivariate surveys. *Survey methodology 15*, 1 (1989), 47–57.

[7] BRANDT, S., MALTZAHN, C., POLYZOTIS, N., AND TAN, W.-C. Fusing data management services with file systems. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage (PDSW '09)* (New York, NY, USA, 2009), ACM, pp. 42–46.

[8] CALLAN, J., AND CONNELL, M. Query-based sampling of text databases. *ACM Trans. Inf. Syst. 19* (April 2001), 97–130.

[9] CAUSEY, B. Computational aspects of optimal allocation in multivariate stratified sampling. *SIAM Journal on Scientific and Statistical Computing 4* (1983), 322.

[10] CHAUDHURI, S., DAS, G., AND NARASAYYA, V. Optimized stratified sampling for approximate query processing. *ACM Transactions on Database Systems (TODS) 32*, 2 (2007), 9.

[11] CHAUDHURI, S., DAS, G., AND SRIVASTAVA, U. Effective use of block-level sampling in statistics estimation. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data* (2004), ACM, p. 298.

[12] CHROMY, J. Design optimization with multiple objectives. In *Proceedings on the Research Methods of the American Statistical Association* (1987), pp. 194–199.

[13] CNET. Security guide to customs-proofing your laptop. *http://news.cnet.com/8301-13578_3-9892897-38.html* (2009).

[14] COCHRAN, W. Sampling technique. *New York: John Willey & Sons* (1977).

[15] DAS, G. Survey of approximate query processing techniques (tutorial). In *International Conference on Scientific and Statistical Database Management (SSDBM '03)* (2003).

[16] DASGUPTA, A., DAS, G., AND MANNILA, H. A random walk approach to sampling hidden databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD '07)* (2007), pp. 629–640.

[17] DASGUPTA, A., JIN, X., JEWELL, B., ZHANG, N., AND DAS, G. Unbiased estimation of size and other aggregates over hidden web databases. In *Proceedings of the 2010 international conference on Management of data (SIGMOD)* (2010), pp. 855–866.

[18] DASGUPTA, A., ZHANG, N., AND DAS, G. Leveraging count information in sampling hidden databases. In *Proceedings of the 2009 IEEE International Conference on Data Engineering* (2009), pp. 329–340.

[19] DASGUPTA, A., ZHANG, N., DAS, G., AND CHAUDHURI, S. Privacy preservation of aggregates in hidden databases: why and how? In *Proceedings of the 35th SIGMOD international conference on Management of data* (2009), pp. 153–164.

[20] DAVID, H. A., AND NAGARAJA, H. N. *Order Statistics (3rd Edition)*. Wiley, New Jersey, 2003.

[21] ELLARD, D., LEDLIE, J., MALKANI, P., AND SELTZER, M. Passive nfs tracing of email and research workloads. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)* (Berkeley, CA, USA, 2003), USENIX Association, pp. 203–216.

[22] GAROFALAKIS, M. N., AND GIBBON, P. B. Approximate query processing: Taming the terabytes. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)* (2001).

[23] GOOGLE. Google desktop. *http://desktop.google.com/*.

[24] HEDLEY, Y. L., YOUNAS, M., JAMES, A., AND SANDERSON, M. A two-phase sampling technique for information extraction from hidden web databases. In *Proceedings of the 6th annual ACM international workshop on Web information and data management (WIDM '04)* (2004), pp. 1–8.

[25] HEDLEY, Y.-L., YOUNAS, M., JAMES, A. E., AND SANDERSON, M. Sampling, information extraction and summarisation of hidden web databases. *Data and Knowledge Engineering 59*, 2 (2006), 213–230.

[26] HUA, Y., JIANG, H., ZHU, Y., FENG, D., AND TIAN, L. SmartStore: a new metadata organization paradigm with semantic-awareness for next-generation file systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)* (2009), ACM, pp. 1–12.

[27] ILYAS, I. F., BESKALES, G., AND SOLIMAN, M. A. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys 40*, 4 (2008), 1–58.

[28] IPEIROTIS, P. G., AND GRAVANO, L. Distributed search over the hidden web: hierarchical database sampling and selection. In *Proceedings of the 28th international conference on Very Large Data Bases (VLDB '02)* (2002), pp. 394–405.

[29] KOGGE, P., BERGMAN, K., BORKAR, S., CAMPBELL, D., CARLSON, W., DALLY, W., DENNEAU, M., FRANZON, P., HARROD, W., HILL, K., ET AL. Exascale computing study: technology challenges in achieving exascale systems. *DARPA Information Processing Techniques Office 28* (2008).

[30] LEUNG, A. Organizing, indexing, and searching large-scale file systems. Tech. Rep. UCSC-SSRC-09-09, University of California, Santa Cruz, Dec. 2009.

[31] LEUNG, A., ADAMS, I., AND MILLER, E. Magellan: a searchable metadata architecture for large-scale file systems. Tech. Rep. UCSC-SSRC-09-07, University of California, Santa Cruz, Nov. 2009.

[32] LEUNG, A. W., SHAO, M., BISSON, T., PASUPATHY, S., AND MILLER, E. L. Spyglass: fast, scalable metadata search for large-scale storage systems. In *Proccedings of the 7th conference on File and Storage Technologies (FAST)* (Berkeley, CA, USA, 2009), USENIX Association, pp. 153–166.

[33] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proccedings of the 7th conference on File and Storage Technologies (FAST)* (Berkeley, CA, USA, 2009), USENIX Association, pp. 111–123.

[34] LOHR, S. Sampling: design and analysis. *Pacific Grove* (1999).

[35] LYNN, W. J. Defending a new domain: the pentagon's cyberstrategy. *Foreign Affairs* (September/October 2010).

[36] MURPHY, N., TONKELOWITZ, M., AND VERNAL, M. The design and implementation of the database file system. *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.11.8068*.

[37] NUNEZ, J. High end computing file system and IO R&D gaps roadmap. *HEC FSIO R&D Conference* (Aug. 2008).

[38] OLKEN, F., AND ROTEM, D. Simple random sampling from relational databases. In *Proceedings of the 12th International Conference on Very Large Data Bases* (1986), pp. 160–169.

[39] OLKEN, F., AND ROTEM, D. Random sampling from database files: a survey. In *Proceedings of the fifth international conference on Statistical and scientific database management* (1990), Springer-Verlag New York, Inc., pp. 92–111.

[40] OLSON, M. The design and implementation of the Inversion file system. In *Proceedings of the Winter 1993 USENIX Technical Conference* (1993), pp. 205–217.

[41] PIKE, R., PRESOTTO, D., DORWARD, S., FLANDRENA, B., THOMPSON, K., TRICKEY, H., AND WINTERBOTTOM, P. Plan 9 from bell labs. *Computing systems 8*, 3 (1995), 221–254.

[42] PLAN 9 FILE SYSTEM TRACES. *http://pdos.csail.mit.edu/p9trace/*.

[43] SELTZER, M., AND MURPHY, N. Hierarchical file systems are dead. In *Proceedings of the 12th conference on Hot topics in Operating Systems (HotOS '09)* (2009), pp. 1–1.

[44] SLASHDOT. Laptops can be searched at the border. *http://yro.slashdot.org/article.pl?sid=08/04/22/1733251* (2008).

[45] SNIA. NFS traces. *http://iotta.snia.org/traces/list/NFS* (2010).

[46] STAHLBERG, P., MIKLAU, G., AND LEVINE, B. N. Threats to privacy in the forensic analysis of database systems. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD '07)* (New York, NY, USA, 2007), ACM, pp. 91–102.

[47] SZALAY, A. New challenges in petascale scientific databases. In *Proceedings of the 20th international conference on Scientific and Statistical Database Management (SSDBM '08)* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 1–1.

[48] VITTER, J. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS) 11*, 1 (1985), 57.

[49] ZHU, Y., JIANG, H., WANG, J., AND XIAN, F. HBA: Distributed Metadata Management for Large Cluster-Based Storage Systems. *IEEE Transactions on Parallel and Distributed Systems 19* (2008), 750–763.