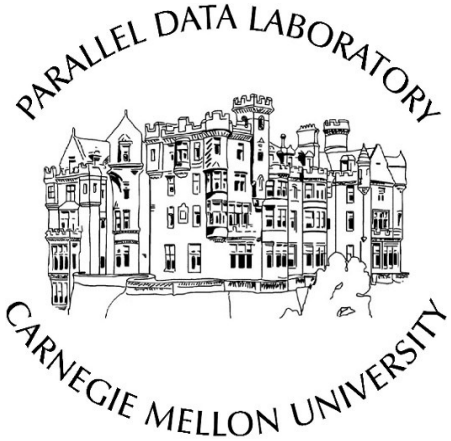


SmartScan: Efficient Metadata Crawl for Storage Management Metadata Querying in Large File Systems

Likun Liu*, Lianghong Xu†, Yongwei Wu*, Guangwen Yang*, Gregory R. Ganger†
*Tsinghua University, †Carnegie Mellon University

CMU-PDL-10-112
Oct 2010

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

SmartScan is a metadata crawl tool that exploits patterns in metadata changes to significantly improve the efficiency of support for file-system-wide metadata querying, which is an important tool for administrators. In most environments, support for metadata queries is provided by databases populated and refreshed by calling stat() on every file in the file system. For large file systems, where such storage management tools are most needed, it can take many hours to complete each scan, even if only a small percentage of the files have changed. To address this issue, we identify patterns in metadata changes that can be exploited to restrict scanning to the small subsets of directories that have recently had modified files or that have high variation in file change times. Experiments with using SmartScan on production file systems show that exploiting metadata change patterns can reduce the time needed to refresh the metadata database by one or two orders with minimal loss of freshness.

Acknowledgements: We thank Michael Stroucken, Raja Sambasivan, and Michelle Mazurek for their comments nd suggestions on this paper; Mitch Franzos, Wusheng Zhang, Qiaoke Zhang and Ze Chen for helping test our scripts; and our colleagues in Parallel Data Laboratory for their early feedback and discussions on this work.
We thank the members and companies of the PDL Consortium (including APC, EMC, Facebook, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, NEC, NetApp, Oracle, Riverbed, Samsung, Seagate, STEC, Symantec, VMware, and Yahoo!) for their interest, insights, feedback, and support. We also thank Intel and NetApp for hardware donations that enabled this work. This material is based on research sponsored in part by CyLab at Carnegie Mellon University under grant DAAD19–02–1–0389 from the Army Research Office, by the Natural Science Foundation of China via project #60963005, National High-Tech R&D (863) Program of China via project #2009AA01A134 and the National Basic Research (973) Program of China via project #2007CB310900.

Keywords: Metadata Crawl, Storage Management, File Systems

1 Introduction

File-system-wide metadata querying is an important tool for administrators that manage large-scale file systems, providing information to guide space management, charge-back, and provisioning decisions. Unlike desktop search, which is designed to assist users in locating specific files, such metadata querying is mainly used by administrators to monitor and better understand the overall make-up of their file systems. It is used to answer questions like "which directory subtrees consume the most space?" or "which files have gone unused the longest?" (making them good candidates for migration to a lower tier of storage).

Most file systems provide no native support for metadata querying. So, this functionality is generally implemented by external tools that maintain a separate database with a copy of the file system's metadata. The primary challenge in supporting efficient metadata querying is the extraction of the raw metadata from the file system to populate that database, which generally requires reading the metadata (e.g., via stat() calls) for each file, one by one—we refer the extraction process as a metadata crawl.

For the large file systems that most need such storage management tools, a single full-speed crawl can take many hours or even days, due to disk seeks and (in the case of network file systems) communication; in addition to the delay, this crawling activity can interfere with real applications' use of the storage system. Worse, keeping the database up-to-date requires regular crawling, currently involving a stat() of every file to see if its metadata has changed. As an example, we have observed a commercial system taking more than seven hours for a single crawl of a 1.6TB file system with 19 millions files; similar observations are also reported by other researchers [9]. Given that truly large file systems may contain hundreds of millions, or even billions, of files, frequent full crawling of such file systems is impractical.

This paper describes approaches to minimizing the subset of files scanned. An ideal crawl would scan only those files whose metadata has changed since the last crawl. Doing so would significantly reduce the work involved with crawling, since the percentage of files changed any given day is small in most file systems [1, 4]. Unfortunately, knowing which files are modified without actually doing a full scan is difficult, since examining values returned via stat() is the standard way of determining whether a given file has changed. So, we focus on approaches to predicting which files need to be scanned.

As with many other aspects of computer systems, metadata changes exhibit high temporal and spatial locality. Our analysis of change patterns in some real file systems shows that directories with files that changed recently will likely have changed files again—directories where work has been ongoing tend to remain active. We also observe that directories with high variation in file change times often have otherwise difficult-to-predict changes—these directories represent locations of relatively frequent but not immediately local-in-time intermittent activity (e.g., periodic backups or new user directory creation). Data about such patterns from two production file systems are presented. Such patterns in how metadata changes can be exploited to reduce the subset of all files that must be scanned to refresh the database.

An inherent danger of predicting a limited subset of files to scan during a refresh crawl is that some changes may not be reflected in the "updated" database. Fortunately, most uses of metadata querying for storage management can tolerate "out-of-date" information to some extent. For example, providing a slightly misordered list of the least active subtrees or slightly imprecise values for the capacity used by each is not unacceptable. So, as long as the predictions of which files to crawl are reasonably complete, the resulting metadata query results will usually be sufficient. When it's not, a full crawl is needed.

This paper describes SmartScan, a metadata crawler prototype that utilizes the new change prediction schemes outlined above. SmartScan uses only POSIX standard file system interfaces (stat(), in particular) to extract metadata, allowing it to be used with any POSIX-compliant file system. Experiments with using SmartScan on two production file services show the desired benefits: (1) restricting the crawl to just the files in directories predicted to contain changed files reduces crawling work by one to two orders of magnitude, and (2) the lost freshness is minimal, and results for our sample metadata queries deviate only minimally from full-scan values.

The remainder of this paper is organized as follows. Section 2 discusses key metadata crawl issues and related work. Section 3 identifies some metadata change properties that can be exploited. Section 4 describes the design of SmartScan. Section 5 evaluates SmartScan's effect on the work of metadata crawling and the resulting freshness.

2 Background and related work

This section describes metadata crawling, associated challenges, and related work.

2.1 Metadata crawl

Metadata crawl denotes the process of extracting raw metadata from a file system and porting it into a seperate metadata database. File system metadata includes both file system native attributes such as inode fields (e.g. size, owner, timestamp, etc.), generated by the storage system itself, and extended attributes (e.g., document title, retention policy, backup dates, etc.), specified by users and applications. In this paper, we only focus on crawling file system native attributes as they are generally considered to be more important to storage management.

Metadata crawl is different from the traditional web crawl in many ways. First, hierarchical file systems have more restrictions than web pages linked by links. Second, web crawl is a best-effort crawl, trying to crawl as many web pages as possible since it is infeasible (for most) to crawl all web pages on the Internet. Nevertheless, with metadata crawl, we are intuitively asking for all the metadata, since global statistics are much more important in managing storage systems (e.g. global space consumption for capacity provisioning). Third, Scaling out is more difficult in metadata search. It is easy to collect web pages by parallelizing crawl processes. But, for metadata crawl, administrators need to make sure the crawl process does not interfere too much with the normal file system workload. It is easy to overload the file service by launching a bunch of processes to crawl the metadata in parallel. Besides, once the disk bottleneck is encountered, increasing parallelism helps little.

2.2 The metadata crawl problem

Metadata crawling is challenges for three primary reasons: First, disk is still the dominant storage medium for large file systems. Yet, trends in disk technology and file systems have made and continue to make metadata crawling a time-consuming activity: first, magnetic disk capacity grows faster than the mean file size. For example, Kryder's law[15] predicts that the storage capacity of a single hard disk doubles annually (increases exponentially); yet, the mean file size grows roughly by 15% per year[1]. This means there will be more and more files per disk as time goes by. Second, disk capacity grows faster than data access performance. Nowadays, even expensive high end server drives can only perform hundreds of seeks per second, having improved only minimally over the last ten years, compared to the capacity. This means scanning even a single disk is becoming an increasingly time consuming task.

Second, the hierarchical structure and standard interface of modern file systems are inefficient for metadata crawl. The iteration-based interface forces an individual "*stat*" operation on each file. Almost all the file systems do some kind of co-location of metadata and data to improve data access performance. Although it helps reduce expensive disk seeks greatly when reading both data and metadata together, it scatters metadata across the disk, leading to more seeks when only doing metadata crawl.

Third, "*stat*" over network adds overhead and possibly another bottleneck. Large file systems are usually server-based and may be distributed across multiple nodes. As an example, we have found that a high-performance commercial NFS server over a 1Gbps network gives us only 1000 *stat*s per second.

"*readdirplus*" can mitigate this problem by pre-fetching inode information for the entire directory, doesn't help much due to the small size of most directories.

2.3 Current metadata crawl approaches

Several solutions exist for collecting metadata from file systems and synchronizing the copies, among which the most widely adopted approach is a periodical scan (e.g. used by *locate*, *GNU Disk Use Analyzer* etc), because: (1) It is the most straightforward solution. It is fast enough on a desktop environments, usually taking only several minutes, and portable since it requires no operating system specific mechanism but the standard file system interface. (2) More importantly, it is so robust that even other neat approaches may still need a periodical scan to ensure correctness. However, the major drawback with this approach is, that it is too slow to be efficient for a large scale storage system. It may take days or even weeks to finish a full scan due to the huge number of files contained and all kinds of overhead in large file systems. Parallel scanning could be employed to accelerate the scanning process, but may consume too much system resource, interfere with normal file system operations, and thus seriously hurt performance.

Another alternative used mostly by desktop searching services is synchronizing the metadata copy based on the file system event mechanism (e.g. "*Change notification*" on Windows, "*inotify*" in Linux[12] etc). These mechanisms enable applications to know exactly when and which files have been changed. Several reasons make these mechanisms not practical in large scale storage systems: First, the large runtime overhead. For example, inotify on Linux requires all watched inodes to be kept in memory, placing a heavy burden on the kernel memory subsystem (Considering a file server with millions of directories). Additionally, as the number of monitored files increases, the notification efficiency decreases for traversing too many monitor entries. That's why inotify has a configurable limit (8192 by default) on the number of watches. Second, this approach doesn't work over network file system. Last but not least, updating metadata copy on every change is inefficient because most of the metadata modifications are overwritten or canceled by subsequent operations. For example, overwriting happens when the write operation cause the file size to increase, or cancellation happens when newly created files are deleted.

The above limitations of periodical scan and file-system-event-based refreshing have driven some research to explore metadata crawl and synchronization via non-standard interfaces. Snapshot-based metadata crawl is proposed in spyglass[9] to capture the modifications of file systems. Although claimed to be much more efficient than a full scan, unfortunately, it is not portable. Its efficiency is tightly coupled to the sequential-read of inode file, and the recursive copy-on-write mechanism, both of which are specific to WAFL[8] and can not be used with most other file systems. Besides, even if it could capture changed inodes efficiently, mapping an inode number to the canonical path may involve extra work that may be expensive.

Changelog, such as the one used in NTFS, is another alternative. However, reading the log without using the file system interfaces is dangerous and not portable even on different versions of the same type of file system. Also, this approach may require hooks into critical code paths, potentially impacting performance and stability. Even if the source code is available to modify, people are usually reluctant to do that due to reliability and stability concerns.

Although there is no existing general solution to efficiently collecting metadata and synchronizing it in large scale storage systems, we believe only crawling the changes is a promising approach. Also, we are arguing, by this paper, that file systems provide enough hints on how files are changed, even by only using information from the standard file attributes. We propose SmartScan, a novel approach to synchronize the metadata copy in management oriented metadata search. SmartScan tries to capture the metadata changes based on the information of standard file system metadata using standard file system interfaces.

3 Understanding metadata changes

The key issue to crawl only the file system changes is to know how a file system changes over time. Understanding metadata changes in large-scale storage systems is crucial not only to efficient metadata collection, but also to partitioning and building change-aware indices.

To better understand file system change properties, we measured metadata changes on several large file systems in the real world, and then used the results to guide the design of SmartScan. We found several key properties in these studies: (1) file systems change little even on large scale storage systems; (2) metadata changes present both spatial locality and temporal locality; (3) DTSD can be used to predict file system changes.

3.1 Data collection

To collect the statistics of file system changes we modified *fsstats*[5], a perl tool that runs through a file system and creates statistics on file attributes such as size, capacity, ages, directory size and etc. We add codes to enable *fsstats* to record file system changes in last day, week, and month, to calculate the change ratio, change locality and commonality, and DTSD. See the following subsections for exact definitions of all these metrics.

To measure the file system changes,we use *mtime* and *ctime*. We choose to ignore *atime*, because: (1) Talking with some storage administrators shows us that *atime* is not considered as important as *mtime* and *ctime* in real storage management tasks. Some file systems don't even keep track of *atime*. (2) Even for those file systems that do support *atime*, many people choose to turn off atime tracking by setting *noatime* option when mounting them to trade for considerable performance boosting. Tracking *atime* according to POSIX semantic forces an inode update for each read which will be a huge overhead. (3) *Atime* is also too volatile to be useful. Many applications have side effects which will change *atime*, e.g. some old backup tools, anti-virus software, etc. We prefer *ctime* than atime because ctime tends to give more reliable results for lack of standard interfaces to change it.

We collect data from a storage server (ServerA) used by more than 100 university researchers for code development, paper writing and serving data for various simulations and data analysis; and a file server (ServerB) used by more than 3000 student at a different university for personal data backup and online sharing. Detailed information about both file systems is provided in Section 5.1.

For clarity of description, we define the following terms:

- **changed directory** A directory at least one of whose children (including both files and its sub directories) is changed during the specified period (e.g. last day).
- **changed file** A regular file whose metadata is changed during the specified period.
- **changed directory file** A directory whose metadata is changed.
- **changed entry** Both changed file and changed directory file are considered as changed entries.
- **directory timestamp standard deviation (DTSD)** A directory's DTSD is the standard deviation of its children's timestamps.

3.2 Categorization of file system changes

It seems almost certain that file systems never change arbitrarily due to its inherent nature by design. From different perspectives, we can categorize the changes using different approaches.

Based on how often files are modified by users or applications, two common access patterns exist: (1) frequent change, that is, files are changed by the users or applications frequently. For example, a developer may compile his project every few minutes, a student may modify his paper every day, or an application may write its operation logs every minute. (2) occasional change, that is, files are changed by users or applications occasionally and can sit untouched for a long time. For example, a user may install a software package, archive a document, download some music, etc.

Based on how a file system is changed, two common change patterns exist: (1) batch changes, that is, the entire directory or subtree may be changed within a short moment. For example, importing photos from a camera, compiling a project, installing a software package, and inflating a compression package. (2) individual change, that is, only one or several files in a directory or subtree may be changed individually. For example, editing a file, downloading a file, or archiving some files.

Intuitively, it is easier to predict the frequent changes, but much harder to capture the occasional changes. However, capturing occasional changes can be very important in some circumstances even if they are individual changes, because most of the occasional batch changes are usually clustered in a subtree whose root is individual changes (e.g. installing software package, archiving by copy to backup directory). As a result, missing such occasional individual changes may hurt freshness.

3.3 File systems change little

As expected, the amount of files changed every day,week and month, even in large files systems, is a small portion. To describe how much of a file system is changed, we use "change ratio" which is defined as the number of changed entries over the number of total entries in a file system. Table1 lists the mean change ratio measured by both ctime and mtime for the file systems we studied.

FS	Last day	Last weekly	Last monthly
A ctime	0.07%(0.0003)	0.34%(0.003)	1.33%(0.004)
A mtime	0.07%(0.0003)	0.33%(0.002)	1.08%(0.004)
B ctime	0.04%(0.0008)	0.44%(0.002)	3.12%(0.041)
B mtime	0.04%(0.0008)	0.44%(0.002)	3.11%(0.041)

Table 1: The average change ratios of ServerA and ServerB, with the standard deviations shown in parentheses.

According to table 1, the change ratios of both file systems we studied are less than 1% per week. This is because both file systems contain lots of files that would almost never be changed after creation. For example, software distributions, photos, musics and videos, archives, etc. Similar results are also reported by other studies via file age statistics [1, 4]. This is a very promising result, which implies that scanning only the file system changes will possibly be much faster.

3.4 Spacial locality of metadata changes

Metadata changes exhibit apparent spacial locality, which means that changes are clustered in the namespace (i.e. occurring in relatively few directories, not scatted evenly across the namespace). Spacial locality comes from the way users and applications organize their files in file systems. For example, users tend to group files related to a particular task into a single directory; some applications (such as photo manager) are designed to use specific directories or subtrees to store their data. Files in the same group tend to change together, e.g. compiling a project, importing a bunch of photos into album or installing a software package, leading to the changes aggregated within a small number of directories or subtrees.

Figure 1 illustrates the worst cases (considering spacial locality) of the distribution for entries that changed in last day and week accumulated by subtree for the ServerA and ServerB. The changes is measured

by ctime during the measured period, but mtime has almost the same distribution. We consider a directory and its parent in the same changed subtree if and only if (1) both the directory and its parent are changed directories and (2) the age of the directory is less than the measured period. This definition allows to capture all the changes under a subtree by checking only its root.

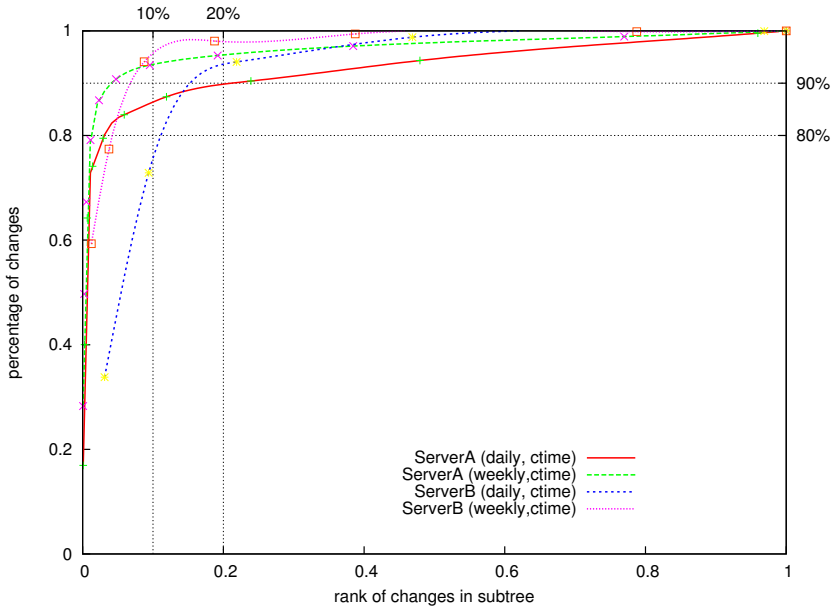


Figure 1: **Distribution of changes measured by ctime**. A rank of 1 represents the subtree with the highest number of changes. A 0.2 of x value means the top 20% of most frequently changed subtrees.

From figure 1 we can clearly see how different types of change patterns contribute to the overall file system changes. The fact that more than 90% of changes concentrate in less than 20% subtrees demonstrates that file system changes are dominated by batch changes localized within a small part of the namespace. However, a long tail indicates the number of changed directories is mainly contributed by individual changes. Again, Capturing individual changes is important since most of the occasional batch changes are usually clustered in some subtrees whose roots are individual changes. Utilizing spacial locality can help us prune the crawling space by identifying only the parts of the namespace where changes happen.

3.5 Temporal locality of metadata changes

Temporal locality means that entries changed at one point of time are likely to be changed again sometime in the near future. Temporal locality comes from how users use file systems every day. Usually, a user works on a few specific projects during a relative long period before switching to others, leading to some hot working sets that change frequently.

However, quantifying the commonality between each day's changes is not trivial. Intuitively, we can use the average percentage of common changes against the total changes as a metric. Practically, sometimes stuffing a large directory tree into a file system can screw the measurement significantly. Rather than letting one number represent all, we visualize changes between each day using two metrics: commonality ratio and coverage ratio. *Commonality ratio* is defined as the common changes between two consecutive days as the percentage of the total changes in the first day. It shows how many changed entries would possibly change again during one day. *Coverage ratio* is defined as the number of common changes over the total changes in the next day. It indicates that the number of changes in one day can be inferred by crawling only the changes

in the previous day. For commonality ratio we measure changes in terms of both directories and subtrees, whereas for coverage ratio, we only measure subtree changes in order to avoid the impact of stuffing large directory tree. Figure 2 shows the commonality ratio and coverage ratio respectively for ServerA based on ctime (again mtime represents very similar result).

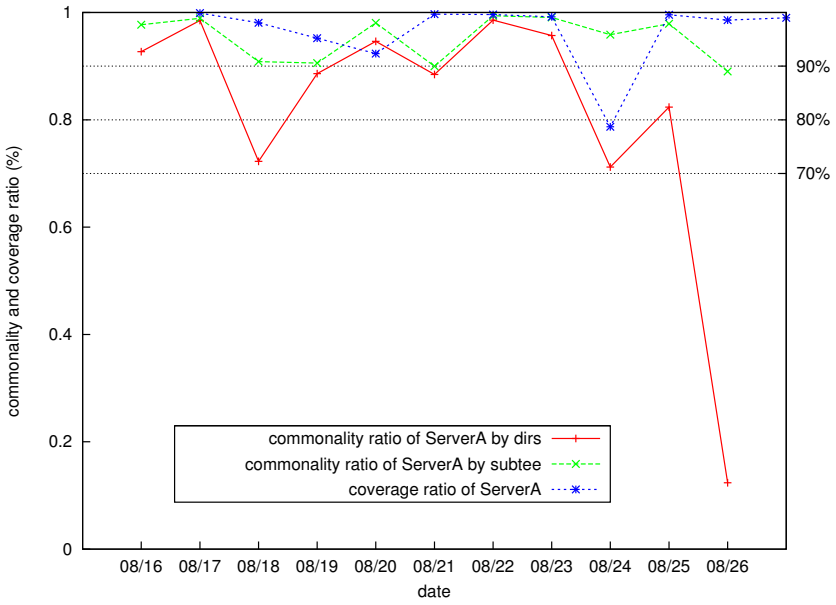


Figure 2: Commonality ratio and coverage ratio

Most of the time, commonality ratio by directory is greater than 70%, meaning that more than two thirds of directories changed one day are going to change again the next day. Although unusual, we do observe a commonality ratio less than 20%, which is caused by an user importing a big directory into the file system. By contrast, commonality ratio by subtree is even higher, indicating that although differences between each days changes exist, they tend to reside in the same changed subtree. The commonality ratio by subtree changes more smoothly, meaning that the pike are caused by changes localized to a few subtrees.In most cases, coverage ratio is greater than 90% means that even only scanning the directories changed in last day could capture more than 90% of the changes for most of the time. Interestingly, although we have a really low commonality ratio for date Aug. 26th, we still have a very high coverage ratio because the roots of those changed subtrees causing the spike are captured by last day's changes.

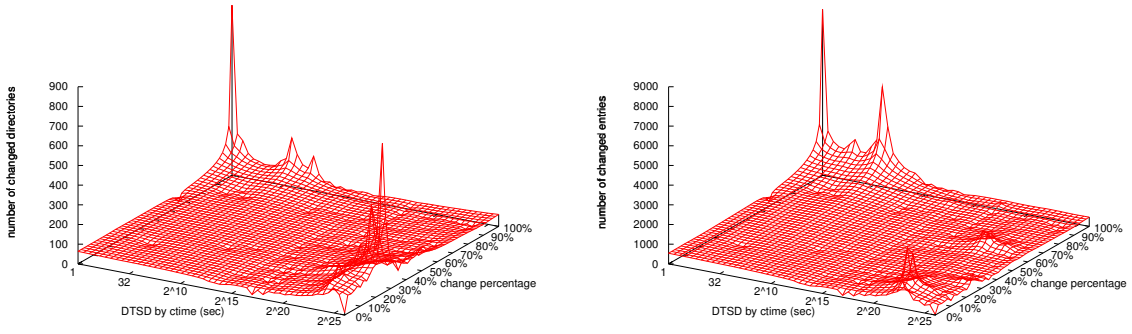
This figure demonstrates that temporal locality provides sufficient hints for changes in home directory workload. However, we don't show the commonality ratio for ServerB because the server is much more fluctuant, since the changes to ServerB are mostly adding new files by different users, in which case temporal locality may not be suitable for change predication of backup workload.

3.6 DTSD sheds light on individual changes

Temporal locality is useful to predict frequent changes, but helps little with occasional changes. Capturing occasional changes is important in some circumstances even if they are individual, because most of the occasional batch changes are usually clustered in subtrees whose roots are individual changes. To better capture occasional changes, we explore another hint from file systems — Directory Timestamp Standard Deviation (DTSD).

Since only part of the directory content is changed when occasional individual changes occur, it is reasonable to expect the changed directory have a higher DTSD. Figure 3 shows the accumulated changes

in terms of both changed entries and changed directories as a function of change percentage and DTSD measured in ctime.



(a) Distribution of changed directory

(b) Distribution of changed entry

Figure 3: The distribution of file system changes in terms of directories and entries as a function of change percentage and DTSD measured in ctime.

The two big peaks with DTSD less than 1024 seconds and change percentage greater than 80% are caused by batch changes. Obviously, this kind of changes account for most of file system changes. Another peak exists in the corner with DTSD between 2^{22} and 2^{24} seconds and change percentage less than 55%, which means most individual changes happen in those directories that have large DTSD. Interestingly, even if there are big peaks in terms of changed directory with changed percentage between 40% and 55%, the surface of changes in terms of changed entries is comparatively flat, implying the changed directories forming these peaks are dominated by small individual changes. These two figures together indicate that DTSD can shed light on the detection of individual changes.

4 The SmartScan System

4.1 Design overview

The basic idea behind SmartScan is borrowed from backup system[2]. Rather than scanning the entire file system every time, SmartScan uses baseline scan and incremental crawl. Baseline scan is a full scan that happens between a relatively long period of time (e.g. every two weeks or one month). Incremental crawl is a partial scan that crawls only the file system changes based on the hints from standard file attributes, and happens much more frequently (e.g. every several hours or one day).

Crawling only file system changes is attractive due to the potential considerable performance improvement. However, to make it viable, the system or administrators must decide which and when directories should be crawled. The core question here is what are the respective roles of the system and the administrators in making this decision. On one hand, it is unrealistic and unwieldy to require administrators to proactively specify what and when directories should be crawled. Intuitively, this work should be done automatically by management oriented metadata search system designed to relax administrators from the storage management burden. On the other hand, no one knows a file system better than its administrators. Increasing administrators' control tends to yield more accurate crawl space while minimizing resource contention with normal file system workload.

Solving this dilemma requires separating the responsibilities of identifying which directories to crawl from determining when and how to do so. In our approach, SmartScan automatically classifies directo-

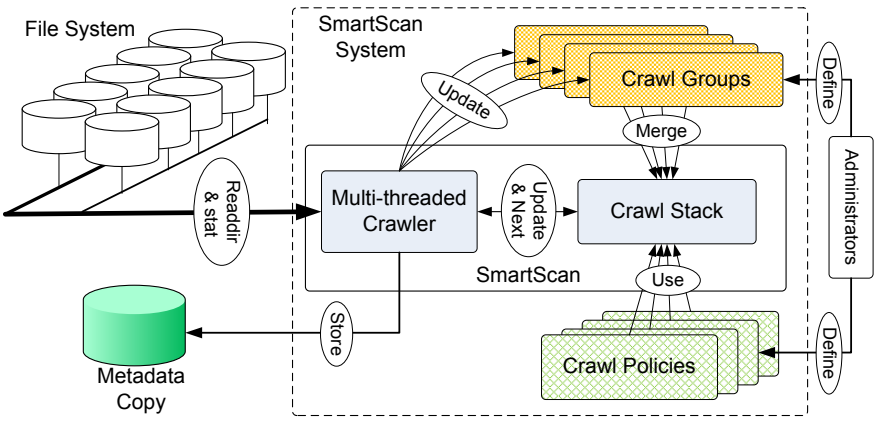


Figure 4: **Architecture of SmartScan**. SmartScan uses only the standard file system interface to gather required information, and thus doesn't require any modification to the file system itself.

ries into crawl groups. The classification process is advisory. Administrators can modify the groups by specifying *include* and *exclude* rules explicitly. SmartScan crawls a file system based on the crawl policy. Administrators' role is to schedule when a crawl should be performed by setting properties for each policy. Since the number of policies is typically small, maintenance of them should be easy enough.

4.2 Architecture

As illustrated in figure 4, SmartScan's architecture is similar to that of most web crawlers. From administrators' perspective, SmartScan mainly consists of three components: a set of crawl groups that define which directories should be crawled, a set of crawl policies that specify when and how to crawl, and the SmartScan program that performs the crawl task. Both crawl groups and policies can be defined or modified by administrators according to their needs. Internally, the SmartScan program has two core components: a multi-threaded crawler and a shared crawl stack. The shared stack rather than a queue is used to maximize the locality impact. Whenever a crawl instance is launched according to a specific policy, it loads the crawl groups specified in the policy, merges them into the crawl stack, and processes them one by one. The merging is required to avoid duplicate entries from different crawl groups. During the crawl, it stores the metadata of each visited entry into an external storage (such as a database or a flat file), updates the crawl groups according to their definitions and adjusts the crawl stack (e.g. when detecting new created directories). Details about crawl groups and policies are provided in Section 4.3 and 4.4.

4.3 Crawl group

File systems themselves provide enough hints on how they change. SmartScan classifies directories into crawl groups based on three file system hints — file age, Directory Timestamp Standard Deviation (DTSD) and subtree size. Groups are adjusted during each crawl to reflect current state of the file system as close as possible. Three specific classification algorithms are used by SmartScan, namely change locality window, DTSD filter, and subtree filter.

Change locality window: The change locality window maintains a list of changed files within the last N seconds. Conceptually, this captures frequent changes happened within a period of time during which a user focuses on a particular task. When the window sees a file whose age (the difference between current time and its ctime or mtime) is less than a specified threshold, namely window size (WS), it considers the file as a change, and creates an entry in the list using the path and timestamp of the file. If such an entry

already exists, its timestamp is updated. The directories to be crawled are implicitly defined by the parents of files in the window.

To generate the directories to be crawled, SmartScan removes those files with age greater than window size first, and then extracts the parent directory for each file left in the window before each crawl. To achieve the best inode cache performance, directories are sorted in nearly lexicographical order, except that a directory and its sub-directories are always sorted together. Smartscan doesn't sort based on strict lexicographical order because, to get better cache performance, directories like */a*, */a.b* and */a/b* should be ordered like */a,/a/b,/a.b* rather than */a,/a.b,/a/b*. The change locality window is updated during both baseline scan and incremental crawl.

DTSD filter: DTSD filter specifies which directories should be considered as crwal candidates where individual changes are likely to happen. The DTSD of each directory encountered in a crawl is calculated. If satisfying the predication of the filter, the directory is added to the group. The DTSD information for each directory in the group is also kept so that it is possible to shrink the group by restricting DTSD value within a smaller range. However, a baseline scan is needed to expand the range. Like groups defined by change locality window, this kind of group is also updated during both baseline scan and incremental crawl.

Subtree filter: The subtree filter splits the entire file system namespace into a set of subtrees with file count above a specified threshold. Each subtree is identified by the path of subtree root and has two attributes: the number of entries under the subtree and the last scan time. Last scan time is necessary when baseline scan is done in stagger mode. This kind of group is only updated during baseline scan. Counting files under a subtree in a multi-threaded program requires more extra work than simply post-order traversing in a single-threaded program. A hash table and reference counters can be used to solve this problem. A reference counter (denoting how many directories it depends on) is kept for each directory encountered in the hash table. Whenever a directory with reference counter 0 is detected, we decrease its parent's reference counter and add its entry number to its parent's. Whenever a directory is detected with file count greater than the specified threshold and reference count equal to 0, it is added to the subtree group.

4.4 Crawl policy

policy	time	mode	groups
hourly	–	seq	–
daily	3:00 a.m., weekday	seq	WS<7 days
weekly	3:00 a.m., Sun	seq	WS<7 days, 16 days<DTSD<256 days
monthly	3:00 a.m., 1st	seq	whole file system

Table 2: **Default policies of the SmartScan**.

To avoid crawl system overloading file systems or interfering too much with normal file system operations, SmartScan uses policy-based crawl scheduling. Policy is a set of rules defining when and which directory groups should be crawled.

A policy is described by three parameters: crawl time, crawl mode and crawl groups. Crawl time specifies when a crawl should be launched, either once or repetitively. Crawl mode indicates which kind of crawl should be performed. Currently, two crawl modes are supported: sequential mode, which crawls each directory in the group sequentially, and stagger mode, which crawls the directories or subtrees not crawled by the last crawl. Last scan time of the subtree is updated after crawling. Crawl groups specify which groups should be crawled. Multiple groups can be specified simultaneously. SmartScan merges directories and subtrees chosen from each group before performing a crawl.

Four kinds of policies are provided by default: crawl hourly, crawl daily, crawl weekly, and crawl monthly. Table 2 summarizes these policies. Administrators can add new policies or modify the existing

File system	description	# of files	# of dirs	capacity
ServerA	Home directories, development, paper writing	16.3 million	1.4 million	1.70 TB
ServerB	Personal data backup and online sharing	3.4 million	0.3 million	15.1 TB

Table 3: **File systems used to evaluate SmartScan**.

policies according to specific needs.

4.5 Implementation

The SmartScan prototype is implemented in Perl on Linux platform. To minimize foreground impact, no daemon process is required to run on the system continuously. Cron[14] is used to schedule when a crawl should be launched. The crawl stack is kept in memory, which works well for our test environment. However, it should not be hard to extend Smartscan to use some external storage such as databases for larger file systems. Each crawl group is kept in a directory with individual files storing directory lists generated and maintained by SmartScan automatically, and rules of inclusion and exclusion defined by administrators. Policies are specified in a configuration file and automatically installed as cron jobs. The POSIX file system interfaces used by the prototype include *opendir(), readdir(), closedir()* and *lstat()*. To avoid overloading the file system, SmartScan allows administrators to specify the number of threads to be launched and the maximum number of *stat()*s per second for each thread. Although SmartScan tries to minimize the modification to file systems, the *atime* of directories may change, depending on specific implementation of atime tracking or the options used when mounting the file systems.

5 Evaluation

Our evaluation of SmartScan consists of three parts. First, we evaluate how many changes we could capture by crawling only those directories that are likely to change. We found out temporal locality is sufficient to capture more than 90% changes, and we could capture almost all the changes with DTSD hints. Second, we evaluate the deviation between query results and real results for ten typical storage management queries. It turns out that the results are satisfactory for most of the queries. Third, we compare the performance of SmartScan with a full scan using both a single thread and multi-threads; as hoped, crawling only file system changes makes SmartScan much more efficient.

5.1 Experimental setup

We evaluate our prototype with two production file systems: a storage appliance (ServerA) used by more than 100 university researchers for code development, paper writing, and data analysis; a file server (ServerB) used by more than 3000 students at a different university for personal data backup and online sharing. The backend store of ServerA is a NetApp filer using WAFL[8] file system with dual parity RAID LUNs created out of 14 SATA disks. The backend store of ServerB is a LVM volume using the XFS file system with RAID5 created out of 10 SATA disks. Although ServerB has more than 3000 users, most of them are inactive. Table 3 summaries both file systems. The performance experiments of ServerB is performed on the file server itself, which is a quad-core Intel Xeon@2GHz machine with 16 GB of main memory running Ubuntu Linux 8.10. For ServerA, because it can only be accessed via NFS, we use another machine with dual-core Intel Xeon@3.00GHz and 2GB of main memory running Debian Linux 5.0.2, to run our prototype and access ServerA.

5.2 Freshness of metadata database

The notion of freshness appears in many contexts, such as database and web indexing. To understand how "up-to-date" a metadata copy is, we borrow the idea of freshness ratio, the most frequently used metric in web crawl, from other work[3]. The freshness ratio, as defined by Cho, indicates how accurate the metadata copy is.

To comprehend the effects of the different hints on capturing metadata changes, we modified our prototype to record not only the captured changes but also the crawl groups to which they belong. We ran the modified prototype on both ServerA and ServerB at 3:00 a.m. during the period from Aug. 23th, 2010 to Aug. 31st, 2010.

Configration	ServerA	ServerB
Unchanged	0.99879 (0.00117)	0.99959 (0.00067)
Daily	0.99994 (0.00010)	0.99983 (0.00023)
Weekly	0.99999 (0.00002)	0.99989 (0.00013)
Daily+DTSD	0.99997 (0.00007)	0.99995 (0.00012)
Weekly+DTSD	0.99999 ($<$ 0.00001)	0.99998 (0.00005)

Table 4: **Freshness ratio of experimental file systems.** *Unchanged* indicates how outdated the copy may get if no crawl is performed; *Daily* and *Weekly* mean crawling only the directories changed in the last day and the last week; *DTSD* means also crawling those directories whose DTSDs are in the specific range ([16, 256] days in the experiments).

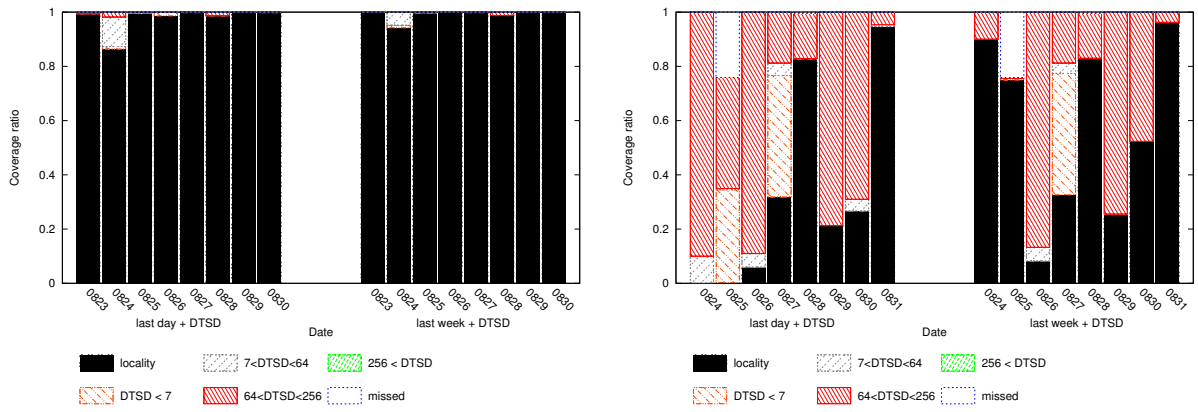
As shown by table 4, the freshness ratios for both file systems, even without performing a crawl, are very good. This result, together with the fact that storage management metadata queries can tolerate outdated data to some extent, implies that very frequent metadata crawl may be unnecessary. With that said, SmartScan does help improve the freshness. The degree of improvement varies when different hints are used.

To show the difference clearly, we redefine the coverage ratio of Section 3.5 in the context of a crawl. Suppose a crawl is finished at time t based on the state of t_0, the coverage ratio is then defined as the number of changes captured by the crawl over the actual number of changes that occurred during this period. The coverage ratio of a crawl tells how many changes that occurred after t_0 could be captured by a crawl finished at t. One thing to note is that, if a *changed directory file* is detected, the directory is also crawled. Figure 5 shows the coverage ratio in terms of ctime for both file systems.

Obviously, temporal-locality-based crawl works really well for ServerA, meaning this approach should be very efficient for home directory workload. Most of the time it is able to capture more than 96% of the changes by only scanning the directories changed last day. Even in the worst case it still captures more than 86% of changes. Nevertheless, it exhibits more diversity with ServerB. It captures almost nothing in some cases, but more than 90% in other cases. This is because ServerB is used as backup storage by many inactive users. The common usage pattern of these users is to put some files into the server every few days. Crawling only the directories changed in last day, or even the last week, is thus not sufficient. However, crawling directories with large DTSD can, as expected, significantly improve the coverage ratio. This is because in such backup workload, users usually classify their files into several categories by putting them into different directories such as documents, music, movies, software, etc., and most of these changes happen in the root of those categories.

5.3 Deviation in metadata query results

Another important metric for out-of-date-ness of a metadata database is the deviation between the query result and the real result, which highly depends on the specific query itself. Unfortunately, there are no



(a) Coverage ratio of ServerA.

(b) Coverage ratio of ServerB.

Figure 5: The coverage ratio indicates how many changes that occurred can be captured by a crawl. "locality" means crawling only the directories changed in the last day or week. $x < DTSD < y$ means the number of changes, among those not covered by locality, that will be captured if we also crawl directories with DTSD between x and y.

standard query sets for metadata search. To evaluate the deviation introduced by selective scanning, we choose ten types of queries considered to be important for storage management. These queries are either derived from our survey of several large storage system administrators and users, or borrowed from other literature [9]. Table 5 summarizes these query sets.

We measure the two most important categories of queries in managing storage systems: aggregation queries and file queries. The result of an aggregation query is a summary value of some specified attributes. We define the deviation of such result set as the absolute difference between query value and the real value over the real value. The result of a file query is a list of files (paths) whose attributes meet the query's predication. The deviation of a file query is defined as the differnece in the number of unique files respectively in the query result set and the real result set over the total number of files in the real result set.

To measure the deviation of each query listed in Table 5, we collect metadata images for ServerA and ServerB by fully scanning the file system during the period from Sep 1st, 2010 to Sep 15th, 2010. We then generate the metadata copies for each day by running a modified version of our prototype against the metadata images using two typical configurations: default configuration for ServerA and weekly locality + $16 < DTSD < 256$ for ServerB. A full scan of the file system takes about 1 hour for ServerA and 5 minutes for ServerB respectively, during which changes may occur. To minimize the impact of such changes, we choose 4:30 a.m., the most inactive period of both file systems, to perform the scan. We consider a change as missing if a less-than-an-hour-future timestamp is detected, even if there is a greater chance that it can be captured by SmartScan. Hence, our result should be considered as an overestimate of deviation.

Different from the previous experiment, new metadata copy is generated by SmartScan based on the last metadata copy rather than the file system image of the previous day, which means a failed detection of a newly created directory will cause all the changes under that directory to be lost—we refer to this as accumulation effect.

For each type of query in table 5, we generate 100 query instances every day. The query parameters is randomly chosen from specific sets that are carefully designed to reflect the real-life storage management queries. We run the query instances on both the metadata images generated by the full scan and the metadata copies maintained by SmartScan. Table 6 summarizes the results.

The average deviations for all ten types of queries are less than 1% (0.01), with standard deviations less

No.	File Management Question	CG	Query Example
1	Which user consumes the most space?	1	Sum size for files using owner
2	Which application files consume the most space?	1	Sum size for files using *extension*
3	How much space does this part of the file system consume?	1	Sum size using *subtree*
4	How many files/inodes does each user have/consume?	1	Count files by owner
5	How many files are there under this subtree?	1	Count files under a *subtree*
6	Which files are consuming the most space?	2	size $>$ *size_v* (e.g. 2G)
7	Which are the biggest files for a specified user?	2	ower=*uid* and size $>$ *size_v*
8	Which files are largest under a specified directory?	2	size $>$ *size_v* and path starts with *path_v*
9	Where are recently modified files by a user?	2	owner=uid and mtime $<$ *mtime_v* (e.g. 2 days)
10	Which files can be migrated to second tier storage?	2	size $>$ *size_v* and atime $>$ *atime_v* (e.g. 6 months)

Table 5: Query Sets. A summary of the searches used to evaluate the deviation of SmartScan. *CG 1* denotes aggregation queries. *CG 2* denotes file queries. *Extension* is chosen from top 200 frequent extensions of the metadata; *subtree* and *path_v* are chosen from those directories who's depth is less than 4; *size_v* is randomly chosen from file size value great than 1MB in the metadata image; *mtime_v* is chosen from one month; and *atime_v* is chosen from 60 days to the max file age.

No.	ServerA				ServerB			
	Accurate	Acceptable	AVG (SD)	Max	Accurate	Acceptable	AVG (SD)	Max
1	98.7%	99.7%	5.3×10^{-4} (0.01)	0.38	99.8%	99.9%	5.7×10^{-4} (0.02)	1.00
2	91.5%	97.8%	3.2×10^{-3} (0.04)	1.30	97.4%	99.9%	5.0×10^{-5} ($<$ 0.01)	0.05
3	99.8%	100%	7.3×10^{-8} ($<$ 0.01)	$<$ 0.01	99.9%	100%	5.0×10^{-4} (0.02)	1.00
4	98.1%	99.7%	1.4×10^{-3} (0.04)	1.05	99.9%	100%	7.0×10^{-5} (0.01)	1.00
5	99.8%	99.9%	4.2×10^{-4} (0.01)	0.50	99.9%	100%	4.7×10^{-4} (0.02)	0.95
6	88.5%	100%	5.1×10^{-4} ($<$ 0.01)	0.01	83.7%	100%	1.5×10^{-4} ($<$ 0.01)	$<$ 0.01
7	99.8%	99.8%	2.8×10^{-4} (0.01)	0.17	99.9%	99.9%	4.1×10^{-4} (0.02)	1.00
8	100%	100%	0.0 (0.00)	0.00	99.9%	100%	1.7×10^{-4} (0.01)	1.00
9	97.9%	99.0%	4.4×10^{-3} (0.06)	1.00	99.8%	99.8%	1.5×10^{-3} (0.04)	1.00
10	86.5%	93.6%	1.5×10^{-3} ($<$ 0.01)	0.03	96.5%	100%	1.0×10^{-5} ($<$ 0.01)	0.00

Table 6: Deviation (DV) of ten typical storage management queries. *Accurate* denotes "$DV = 0$", meaning SmartScan's copy can give the same result as the fully-scan copy does. *Acceptable* denotes "'$DV < 1\%$", indicating that the result is acceptable for storage management purposes. *AVG* and *SD* stand for the average value and standard deviation of the deviations of each query type. *Max* shows the maximum value of the deviations.

than 7%. $7/10$ for ServerA and $8/10$ for ServerB types of queries have more than 95% accurate instances , indicating there is a probability higher than 95% that the metadata copy maintained by SmartScan can give the exact answer as a full scan does. If a 1% deviation is acceptable for storage management purpose, SmartScan is able to give a reasonable result for all but the type 2 and type 10 queries for ServerA with a probability higher than 99%. Large deviations (nearly 1.00) do exist for some specific query instances as shown by column *Max*, implying there is still a chance, although tiny, that SmartScan's copy may give totally wrong result.

The mean deviations are high as compared to the proportion of outdated entries in the metadata copies. This is mainly because the queries we chose for storage management purposes tend to involve many files, which increases the probability of encountering a missing change. Besides, the accumulation effect of missing changes helps to some extent. Queries on ServerA copy tend to have high deviations than ServerB, since individual changes are more likely to be missed when only temporal locality information is used. The much lower percentage of acceptable results of type 2 queries for ServerA comes from a failed detection of the deletion a subtree containing more than 562k files from a user's *project* directory, which would have been captured if DTSD information was used. Since using DTSD information may increase the crawl duration as

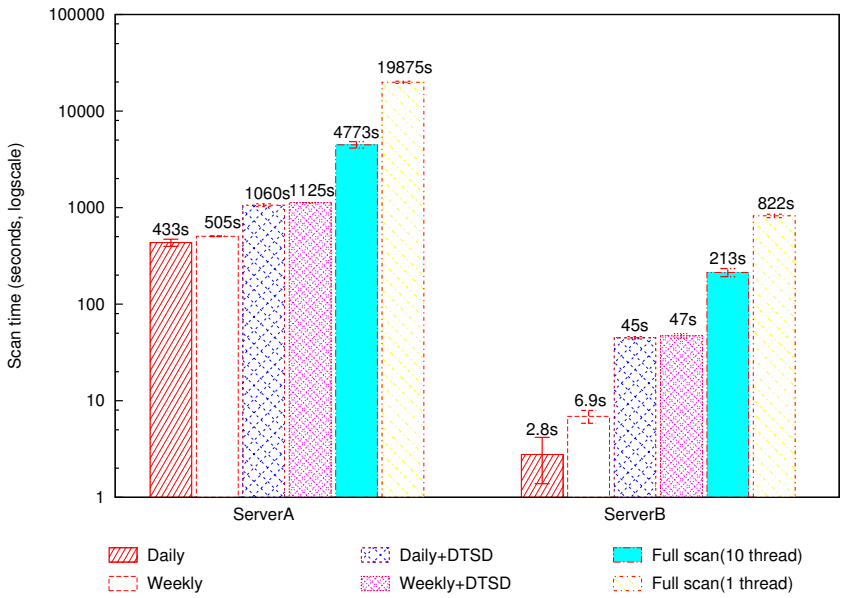


Figure 6: **Crawl performance of four typical configurations.** "Daily" means crawling only the directories changed in last day. "Weekly" is similar. "DTSD" means, besides directories changed in last day or week, directories are also crawled whose DTSD in terms of ctime is between 16 and 256 days.

shown in Section 5.4, it's up to the administrators to decide which policy should be used.

The type 10 queries are somehow interesting because the metadata copy is maintained based on *ctime*, whereas the queries use atime, an attribute whose change will not effect *ctime*. Surprisingly, 93.6% for ServerA and 100% for ServerB of results are acceptable, meaning it is still useful for administrators to obtain a high-level idea about which files should be archived. Again, these results should be considered as an overestimate of the deviation.

5.4 Performance

We evaluate SmartScan's performance by running it with four typical configurations during the period from Aug. 24th, 2010 to Aug. 31th, 2010. We perform the experiments at 4:30 a.m., the most inactive period, to minimize the impact of file system changes. Dentry cache and inode cache have a big impact on the results. Thus, We choose to use a cold cache at the beginning of each experiment so that the results are comparable to each other. In real life environment, SmartScan has a good chance of cache hit, since recently changed directories are likely to be in the cache, so the results here should be considered as an underestimate of the benefits of SmartScan. Two different approaches are used to clear file system caches: we ask the administrator to run "echo 3 >/proc/sys/vm/drop_caches" as root on ServerB; while for ServerA, as there is no interface for us to clear the filer's cache, we try to overwrite the cache by performing a 15-minute partial scan on unrelated subtrees. Both ways prove to be effective according to our observation.

Figure 6 shows the average time needed to crawl both file systems using four typical configurations. All Smartscan's crawls are performed using a single thread. As a comparison, full scans using both one and ten threads are also shown. Depending on the number of changes in the file system, SmartScan can be up to 1-2 orders of magnitude faster than the full scans when using the last day or last week information for change prediction. Even with a crawl exploiting both temporal locality and DTSD hints, SmartScan is still 5× faster than a parallel scan using 10 threads and almost 20× faster than a single-threaded full scan.

Table 7 shows the number of directories to be crawled for the four typical configurations as well as the

Configuration	ServerA	ServerB
Daily	9774.6 (7785.3)	311.0 (299.7)
Weekly	19966.3 (255.7)	992.4 (311.9)
Daily + DTSD	43082.0 (7633.2)	2813.0 (358.1)
Weekly+ DTSD	53003.5 (239.9)	3416.3 (210.9)
Full Scan	1428412.0 (6636.3)	267946.8 (348.7)

Table 7: **The number of directories to be crawled for four typical configurations.** The number of directories for a full scan is also listed as a comparison. References figure 6 for the means of "Daily", "Weekly", and "DTSD". The value listed is mean value of eight days, with the standard deviation shown in parenthesis.

full scans. SmartScan is much more efficient in terms of the number of scanned directories, nearly two orders of magnitude less than a full scan even in the worst case. In contrast, the less improvement in crawl time is mainly caused by the degradation of cache performance when performing selective scan. Further, using cold cache also aggravates the situation to some extent. ServerA is notably slower in terms of stat() per second due to overhead introduced by NFS, showing the impact of network communication on metadata crawl. Using DTSD information on ServerB causes about 3 times increase of the number of scanned directories, but more than 6 times increase of crawl time, because directories added by DTSD hints tend to exhibit poor locality and thus degrade the cache performance. In comparison, the performance overhead of adding DTSD hints on ServerA is not apparent. This is because, on one hand, the network overhead for ServerA alleviates the cache effect; on the other hand, we found, by examining the content of changed directories, that a large directory changed every day exists in ServerA, reducing the impact of DTSD further. As "Daily" in table 7 denotes the number of directories changed every day, all the other policies improve the freshness to some extent at the cost of crawling more useless directories. It's up to administrators to make the tradeoff.

6 Conclusions

Patterns in metadata changes can be exploited to reduce the work involved in refreshing a metadata database used for storage management queries. Further, such selective crawling has minimal impact on freshness and on the results of example metadata queries. Effective predictors of which directories need to be scanned are recent changes in that directory and high variation in constituent file change times. These conclusions are confirmed by measurements of two production file services and by crawling them via SmartScan, a metadata crawl prototype that implements selective scanning.

References

[1] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A five-year study of file-system metadata. *Trans. Storage*, **3**(3):9. ACM.

[2] Ann Chervenak, Vivekanand Vellanki, and Zachary Kurmas. Protecting File Systems: A Survey of Backup Techniques.

[3] Junghoo Cho and Hector Garcia-Molina. Synchronizing a database to improve freshness. *SIGMOD Rec.*, **29**(2):117–128. ACM.

[4] Shobhit Dayal. *Characterizing HEC Storage System at Rest*. Technical report.

[5] Shobhit Dayal, Marc Unangst, and Garth Gibson. Static Survey of File System Statistics. Petascale data storage institute.

[6] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. Pages 59–70. ACM.

[7] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, **37**(5):29–43. ACM.

[8] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. Pages 19–19. USENIX Association.

[9] Andrew W. Leung, Minglong Shao, Timothy Bisson, Shankar Pasupathy, and Ethan L. Miller. Spyglass: fast, scalable metadata search for large-scale storage systems. Pages 153–166. USENIX Association.

[10] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. Pages 4–4. USENIX Association.

[11] M. Satyanarayanan. A study of file sizes and functional lifetimes. Pages 96–108. ACM.

[12] Martin Streicher. Monitor Linux file system events with inotify. IBM Linux Test and Integration Center.

[13] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. File size distribution on UNIX systems: then and now. *SIGOPS Oper. Syst. Rev.*, **40**(1):100–104. ACM.

[14] The Open Group. *The Single UNIX Specification: The Authorized Guide to Version 3*. The Open Group. Open Group Document Number G906.

[15] Chip Walter. Kryder's Law. Scientific American Magazine.

[16] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A. Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable Performance of the Panasas Parallel File System. Pages 17–33, Mary Baker and Erik Riedel, editors. USENIX.