UNIVERSITY OF CALIFORNIA
SANTA CRUZ

IMPROVING FILE MANAGEMENT THROUGH PROVENANCE AND RICH METADATA

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Aleatha Parker-Wood

June 2014

The Dissertation of
Aleatha Parker-Wood is approved:

Darrell D.E. Long, Chair

Ethan L. Miller

Philippe Rigaux

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Improving File Management Through Provenance and Rich Metadata

by

Aleatha Parker-Wood

Modern high end computing systems store hundreds of petabytes of data and have billions of files, as many files as the Internet of only a few years ago. Even modern personal computers store numbers of files that would be massive for the largest mainframe computers of forty years ago. The quantities of data in modern computing have long since overwhelmed anyone's ability to manage it manually, and the forty year old tools currently in use for file finding and management are reaching the limits of scale. In an environment like this, secure, effective, and efficient search algorithms and automatic file management become a necessity, not a nicety.

This dissertation addresses the question of how users can better find and manage files by taking advantage of advances in file systems. We focus on a multi-user scientific computing environment, but many of the techniques we describe are effective and advantageous at desktop scale as well. We begin with an empirical description of the problem, drawn from user studies and our statistical analysis of scientific data, in order to better understand the problem domain. We then describe a new technique for studying provenance in scientific systems, and a technique to synthesize system level provenance from existing traces. We describe our novel algorithm designed to provide importance ranking for file system search by leveraging provenance, and discuss the relationship between ranking and access prediction. And finally, we show how rich metadata can be used to improve file management by automatically generating expressive, unique file names.

Modern file management must be automatic and scalable, allowing users and file systems to focus on what each does best. By exploiting richer information such as provenance and semantic metadata, file systems can offer sophisticated new capabilities to ease the burdens of users, making file systems easier to use, navigate, and understand.

To my daughter, who makes me want to never stop making the world a better place for her.

Acknowledgments

My graduate career would not exist without the help of those who encouraged me to apply to graduate school, and then guided me through. Many thanks to my last advisor, Darrell D.E. Long, and to my first advisor, Ethan L. Miller. Without both of you, I would never have started down this road. You have both listened patiently, shepherded me through the process of learning to research, taught me to communicate clearly, and helped me turn crazy ideas into good ones. You have been fantastic mentors, and I hope to make you proud.

To Philippe Rigaux, thanks for your extraordinary patience in awaiting this long-delayed dissertation, and for hiring me as a post-doc. The VERTIGO group is a superb place to broaden my research horizons, and I'm excited about the research. I hope my contributions will be useful.

To my advancement committee members, Margo Seltzer and Daniel Tunkelang, it has been an honor and a privilege to receive guidance from you, and I hope there will be more chances in the future. This dissertation owes much to you.

My friends, lab mates, and alumni of the SSRC have been a constant source of help, encouragement, and comfort. Everyone there has helped me in some way, but a few specific thanks are due. Many thanks to Stephanie Jones and Christina Strong for endless reviews, matplotlib help, baked goods, board game nights, and true unwavering friendship without which I would have cracked. Mark Walter Storer: thanks, champ. You keep me honest and humble, and I love you for it. Many thanks to Ian Adams and Avani Wildani, and my sincerest apologies for being a stress-monkey. Brian Madden, you're a brilliant mind and a solid dude. I hope to have the privilege of working with you again. Andrew Leung, you got me started on my first projects, and my dissertation topic builds on all the work you've done before me. Thank you for getting the ball rolling. DJ Capelis, you've turned into a fantastic woman and a real lady, and I look forward to arguing with you for the rest of my career. Thanks for all your help debugging, and for lending a shoulder when I needed it. Deepavali Bhagwat, thank you for proving that someone can be a great researcher and a good mother at the same time.

Pour mes estimés collègues de Vertigo, Nicolas Travers, Andrei Stoian, Emilian Pascalau, Marin Ferecatu, Kostas Raftopoulos, Nguyen Vu Hoang, et Michel Cruciano, merci à vous pour tout.

Tom Kroeger and Ahmed Amer, thank you both for listening and guiding me. As you both reminded me, there are only two kinds of dissertations, finished and not finished, and the second one doesn't count. Mark Lillibridge read an early draft of my advancement, and asked some valuable and pointed questions, for which I am very grateful. Meghan Wingate provided access to scientists and data, and has never stopped advocating for better file systems for them both. I hope I've made the world better for your users.

Zachary Mason, you were right about grad school, but I regret nothing. Thanks for looking after my soul.

Cynthia McCarley and Tracie Tucker, for making everything run smoothly and guiding me through the maze of paperwork. Your dedication to your work makes everyone's lives better.

My numerous funding sources and internships, including but not limited to: The National Science Foundation, NASA, IBM, the Naval Post-Graduate School, the Polymathic Project, and the industrial sponsors of the SSRC.

And finally, most importantly, I would like to thank my husband, Aaron Wood, who has dived into every new adventure, from graduate school to children to international moves, with love, trust, and enthusiasm. This dissertation would not exist without him.

The text of this dissertation includes reprint of the following previously published material: Examining Extended and Scientific Metadata for Scalable Index Designs, by A. Parker-Wood, B. Madden, M. McThrow, D. D. E. Long, I. Adams, and A. Wildani, In Proceedings of the 6th International Systems and Storage Conference (SYSTOR 2013), June 2013. I certify that the ideas presented here are mine, but that coauthors listed gave significant assistance with programming, English, and/or corrected my mathematics.

Chapter 1

Introduction and Background

'Begin at the beginning,' the King said gravely, 'and go on till you come to the end: then stop.'

Lewis Carroll

High end scientific computing systems face a crisis. While the computing and storage capacity of computing systems has grown by orders of magnitude, allowing users to create and store ever larger amounts of data, tools for finding and managing data have not kept up. Scientists are still manually naming files, sorting files into directories, and searching by navigating or using brute force tools. Manual file organization leads to millions of files in a single directory, overwhelming the traditional inode structure [84]. Scientists are still using `grep` and `find`, `awk` and `du` to find files and directories which match a criteria such as size or a keyword. These tools are brute force and slow, frustrating the user, and consuming valuable computing resources. Worst of all, the ineffectiveness of manual file organization and naming is resulting in users circumventing the file system all together, and tracking information outside it. Users embed metadata in file names and directories, or use PowerPoint, Excel, Notepad, and three-ring binders to track their files and the relationships between them [93]. Hours of scientists' time are wasted organizing valuable metadata outside the system, where no one else can use it, and computing systems are being overwhelmed by the burden of search requests. This metadata could be managed efficiently and robustly by the file system and used to help users find their files more effectively.

Improving the quality of file management on high end scientific computing systems not only benefits the scientists who use the system, it benefits the system itself, freeing up computing resources for more important tasks. However, a number of advances are needed to make better file management a reality. In this dissertation we describe our key advances in five areas which are designed to improve scientific file management:

- The study of scientific file management from a scientist's perspective (Chapter 3)
- Analyzing scientific metadata (Chapter 4)
- Constructing provenance (Chapter 5)
- File access prediction and ranking using provenance (Chapter 6)
- Automatic file naming using metadata (Chapter 7)

1.1 Studying Scientific Users

To ensure we are solving the correct problems, we engaged in an empirical study of scientific data management, interviewing scientists to discuss how they currently manage their files, and what challenges they face with data management. Our research covers multiple laboratories, and a wide variety of scientific disciplines.

The problems described by scientists include difficulties with finding files, giving files meaningful names that are consistent with the contents of the file, and understanding the lineage of files, along with a number of other problems. In Chapter 3, we describe our user study and its results. This user study is used to motivate and focus our work on file systems.

1.2 Analyzing Scientific Metadata

The scientific world has been vocal about a desire to have richer metadata exposed and readily searchable. While file system metadata is well characterized by a variety of workload studies [19, 37, 38, 68, 100], scientific metadata is much less well understood. Better understanding the sorts of metadata extant in science can allow us to make more intelligent decisions about storage and indexing, allowing us to economize on space and computational power while enabling the rich queries that scientists are interested in.

Current techniques for file system search [50, 67, 75, 97] have been designed and tested for file system metadata, such as POSIX metadata, and fail to account for the wide variety of metadata users would like to search. In Chapter 4, we perform a deep analysis of scientific metadata, characterizing scientific data from several disciplines that might be found on a large system. We consider a variety of aspects that must drive index decisions for scientific data, such as the dimensionality, sparsity, and entropy of the data. We analyze a wide variety of existing solutions to scientific and file system indexing, and offer our conclusions on appropriate storage mechanisms based on what we have learned about the data.

1.3 Constructing Provenance

Provenance is the set of data collected to answer the question “how did this object come to be here?” In the art world, it may describe a chain of ownership and transportation. In computing, especially scientific computing, it refers to questions about data flow, describing the objects and processes which went into creating a given piece of data, such as a file or database entry. We focus specifically on file provenance in this work.

In our study of scientific data management, it became clear that scientists wanted a richer provenance enabled view of the file system, allowing them to track the relationships between different data files, visualizations, presentations, and papers, among many other things. There are a number of systems currently in existence for collecting file provenance, primarily scientific in focus [20, 30, 51, 73, 74]. These fall into two categories: disclosed provenance, where the user or application describes the provenance of a file [20, 30, 51], and observed provenance [73, 74], where the system is responsible for collection. Provenance can be collected automatically, or manually added, and provenance can be collected at a wide variety of semantic levels.

Unfortunately, the majority of existing provenance collection systems did not fill the niche scientists were interested in. The existing systems are designed to serve a specific type of science. They are not broadly applicable, are specific to a certain type of workflow, and are only aware of a small subset of the data within a file system. While these sorts of semantic workflow provenance systems are highly valuable to scientists, they do not cover enough of the file system to enable the kinds of queries scientists need to effectively manage their data.

Our work on data management specifically focuses on automatically collected observed provenance at the system level, which is highly generalizable, and tracks the data flows between files and processes, allowing the kinds of detailed queries users are interested in. This is similar to the type of provenance collected by the Provenance Aware Storage System (PASS) [74]. However, PASS is a full fledged system for collecting provenance which requires a custom kernel, and is highly intrusive, making it difficult to get provenance traces at the system level for any kind of substantial realistic workload. Most available libraries of provenance data are either synthetic in nature, or have poor coverage of a system, focusing only on a high semantic level or collecting data only from instrumented programs.

To design and test our data management algorithms and gain more insight into multi-user file systems, we needed access to a large body of provenance data which did not exist. However, there are several large corpuses of system call traces available. We therefore designed a mechanism for retrospectively constructing observed provenance using existing file system traces, thus allowing us to study data provenance on systems which do not support provenance collection. In Chapter 5, we describe our mechanism for creating provenance data from traces, and discuss the advantages and pitfalls of such an approach.

1.4 Access Prediction and Ranking

Ranked search has a variety of advantages for large file systems. For users, it can improve the user experience by displaying important results early. Ranked search allows the user to issue exploratory queries to investigate the contents of the file system without going through an expensive query process. Research on the web has shown that users who are *re-finding* items that they already know to exist can issue shorter queries which rank the desired item higher than in their initial search [95], which simplifies search for the user. If this also holds true on the file system, it can further save user effort.

Ranked search also has advantages for systems, by reducing space and computational time. By not requiring the system to produce a complete list of results for every query, computing power can be saved at query time, as with top-k results algorithms [110] and index pruning [29, 32, 66, 72]. Additionally, if the user can find their files with the first query, rather than going through a lengthy query refining process, the computing system does not need to retrieve results for multiple queries. Finally, ranking provides a mechanism for determining which files are relevant to queries. Files which are unlikely to be relevant to any query may not be likely to be accessed soon, and so ranking can provide additional guidance for archival and caching policies.

Results ranking is a mature field, with two primary components. The first one, that of *similarity*, is extremely well understood for text content, and is actively researched for other content types such as images and music. These algorithms depend only on the file data itself, and work equally well on the web, databases, and file systems. In this work, we focus instead on the second component, that of *importance*. Importance ranking is designed to rank a large number of files which all match a query well, and might be suitable answers. On the web, this type of ranking is performed by a large number of algorithms, commonly used in ensemble. The most famous of these is PageRank [79], but there are also algorithms which rely on click-through data, linger times, page structure, and many other factors [18, 31, 53]. Judging by the success of web search engines, these algorithms are quite successful indeed. However, ranking for file systems has been much less active. For a long time, the scope of file systems, and the ability of humans to organize the contents themselves, was sufficient. File system search did not need to be good, because file systems were not very large, linear search tools were fast enough, and directories were enough to organize and find data. This no longer holds true. File systems are extremely large. In fact, the largest file systems are within an order of magnitude of the web. Directories are no longer sufficient, both because they are reaching scalability limits [82], and because there are too many files to organize via human effort. Therefore, file system search needs to close the gap, allowing the same ease of use that the web now offers.

To do importance ranking, we need to know what is important. However, that's a difficult question to answer, even for a human. Conventional ranking studies, with human relevance judgements, are hard and expensive, and even experts in an area often will not agree on the importance of a search result [99]. So surrogate metrics on large amounts of activity data are commonly used. File systems capture a different kind of structure from the web, so traditional web ideas of importance cannot be directly applied. Where the web has a rich selection of links between pages to indicate what individuals think is important, file systems have stuck to simple POSIX metadata, such as directory hierarchies, file access time, file creation time, and file system links.

On the web, the primary measures of importance are three-fold: whether you *link* to a page, whether you *click* on a search result, and whether you *linger* on a search result, or immediately return to Google and try a different link. On a file system, by analogy, we might ask the question what do you *open*, and what do you *read*? In other words, on the file system, there is a clear connection between access prediction, and search ranking. If we have an oracle that can predict which files you are most likely to open next, then that oracle can also be used to rank search results, since the best search result will be the one you will open. In Chapter 6, we explore the connection between access prediction and ranking, and describe a hybrid algorithm intended to do both, TaPIR [80].

We demonstrate that we can improve access prediction by leveraging ranking algorithms originally designed for the web, and offer accurate, personalized access prediction by using the rich metadata available from provenance to track relationships between files and processes over time. Our ranking method, TaPIR, uses a time-weighted Markov Chain technique to deliver accurate access predictions, using the graph structure inherent in provenance. To demonstrate its effectiveness, we employ it to do both system-wide access prediction and personalized access prediction.

Provenance access prediction requires more data and computation than many existing access prediction methods, but leverages data that scientists already use in other contexts, and in return, allows a wealth of rich new capabilities. This access prediction can be used to determine file popularity over time, finding recently popular files, and files which are accessed often over a long period. It can also be used to generate an “inverse popularity” ranking, where the user is looking for things which are infrequently used (perhaps looking for underused data sets to explore, or candidates for archival.) In addition, provenance capture allows us to track things on a user by user basis, allowing powerful algorithms for personalization and recommendation, and enabling identification of “working sets” which are often accessed together, as well as providing a new class of useful data for the user to query.

1.5 Metadata Aware Naming

File names have existed since the earliest file systems, and serve two important functions. First, they serve to uniquely identify a file over time. Second, they serve to trigger our memory, describing the contents of a file, and helping us to find it or recognize it when we look at it later. In order to help users to find and remember files, they often contain a bounty of useful metadata about the file.

However, file names, as they currently exist, have numerous flaws:

- They are unstructured semantic metadata which is opaque to the system
- Formatting is up to the user, and is error prone and inconsistent, making it hard to find files later
- Changing a file name can destroy information that is only present in the file name

Consider the following file name drawn from the author’s experiments:
`createfiles_HDD_truenames_100000files_1threads.data`
Looked at in one light, this is a long, arbitrary, error prone string of characters. In another light, it is a rich source of semantic metadata about the contents of the file that could be used for search and analysis, if only we could extract that information.

To resolve these problems, we propose to disassociate the two functions of names, separating the task of uniquely identifying a file from helping users to identify a file. We describe a new file system, TrueNames, designed for both scientists and general users. TrueNames provides a durable unique identifier for a file which can be used by applications, captures rich metadata in a structured format, and uses it to dynamically generate user-friendly file names. These file names have many advantages over conventional file names. They are *correct*, because they are continuously synchronized with the file's metadata. They are *regenerable*, allowing us to compress and recreate names at will without loss of information. File names can be *disambiguated* using all available metadata, which reduces accidental data over-writes. The metadata which we capture is structured, making it readily available for search and data management. We describe these new file names as *metadata aware* names.

TrueNames is designed to meet a number of goals. Metadata aware file naming can make naming more reliable and less error prone. It can also prevent accidental overwriting of existing files by detecting collisions between names, and adding additional metadata to disambiguate. Many applications already offer some form of automatic file naming, and by making that functionality part of the file system, we can speed application development, and prevent code duplication and the resulting proliferation of bugs. Being able to regenerate file names allows us to port files between file systems with differing constraints, generating a meaningful filename in each location, and then reconstructing the original file name whenever needed. We can easily move metadata between file names and directories, or store it for later use. And finally, by gently encouraging users and applications to share more metadata in a structured fashion, we can make metadata more readily available to improve the quality of file system search or support a non-hierarchical file system.

1.6 Summary

The volume of data in modern high end scientific computing file systems has overwhelmed the user's ability to manage it effectively. Searchable file systems fill part of the gap, but we are still left with the problems of presenting the user useful results quickly, and helping the user distinguish between files. The work described in this dissertation is intended to address the question of architecting scientific file systems to improve efficiency, and enabling users to find and manage their files effectively. Our work includes significant advances in our understanding of scientists and scientific data, and provides a tool for creating provenance data sets, along with the largest provenance data flow set available to date. We offer new algorithms for ranking file importance, TaPIR, and managing file names quickly and consistently, TrueNames. Each of these advances the state of the art in scientific data management.

Chapter 2

Related Work

If you don't see the use of it, I certainly won't let you clear it away. Go away and think. Then, when you can come back and tell me that you do see the use of it, I may allow you to destroy it.

GK Chesterton

Our research builds on a broad foundation of previous work, drawing on file system research in metadata, indexing and search, access prediction, and provenance. There is also a large body of work regarding naming, snippets, and faceted search on the web which has guided our research in automatic file naming.

2.1 Data management studies

Many researchers have studied desktop file and data management for general users [25, 26, 58, 60, 103]. Some of these studies have also included the personal files of scientists at universities [28, 59]. However, scientists, particularly in large scale computing, have needs which are slightly different from the general computing community. They must manage much larger volumes of data, and are highly concerned with questions of data flow and provenance. Their files are often shared, rather than personal. In addition, unlike desktop users, who prefer to rely on their own memory [59], scientific users take meticulous steps to record and manage information, making them similar to Malone's *filers*, who explicitly organize information into a pre-defined structure, rather than *pilers*, who throw information into large buckets with no internal structure and then rely on memory to find things again [69]. Scientists often work with highly structured data, and are interested in being able to query it. Ours is the first study to focus exclusively on data management by scientists for scientists, and fills a gap in knowledge about scientific data management.

2.2 File system characteristics and index design

File systems have been heavily analyzed, using collected data to build better file systems and search mechanisms, along with many other improvements. Here we summarize a variety of trace and workload studies, as well as research in choosing indexes for file system search.

2.2.1 Metadata and workload studies

There have been a large number of previous studies which examined file metadata. However, they have focused exclusively on file system metadata and file types, rather than scientific metadata. For instance, Douceur's large-scale study of file-system contents [38], and Agrawal's five-year study of file-system metadata [19], also did detailed statistical analysis of distributions. Both focused on desktop file systems.

On a larger scale, we note Leung's large scale network file system study [68], which tracked behavior and file system metadata for corporate file servers. Perhaps the closest studies to our research on scientific metadata are Dayal's study of HPC at rest [37] and Wang's study of HPC workloads [100]. However, they focused on file system metadata and workloads, where we focus on higher level semantic metadata. Finally, there is Adam's study of scientific archive usage [**?**], which focuses strictly on file system access patterns, sizes, and ages.

2.2.2 File System Indexing

File system indexing is a wide field. Here we focus specifically on systems for searching metadata, rather than text search, since our work is primarily applicable to metadata search. Our work on characterizing scientific metadata is designed to assist scientific system designers in choosing between the available options, and our work on access prediction and ranking pre-supposes the existence of a searchable file system, such as those described here.

Inversion [76] was the first system to propose integrating indexes into the file system, and focused on both system and user-supplied metadata. They proposed replacing the file system with a POSTGRES database, and defining tables for user-supplied data types.

Spyglass [67] focused on how to create indexes with fast update and query time, by using a log and merge technique for index building, multiple indexes with Bloom filters to facilitate early discard of irrelevant areas of the file system, and K-D trees to take advantage of the skewed distributions of metadata. SmartStore [50], rather than using K-D trees, used R-trees and clustering, in an attempt to make index rebalancing more efficient in the face of frequent updates. Loris [97] and Pantheon [75] were both indexing systems for file system metadata. Pantheon used B-trees. Loris used log-structured merge-trees [77]. While these each performed well on their test data, they focused strictly on POSIX metadata.

BeFS [42] was an early desktop system designed to handle both system metadata and extended metadata. In BeFS, all metadata was stored in inodes and resource forks, and then indexed in a B+-tree.

Diamond [52] is a system for large scale search, focused on efficient search where not all searchable attributes can be kept in indexes, and some data must be extracted from files in real time. They describe a system for early discard of files based on filters, using distributed computation to process files and evaluate whether they match a criteria. The techniques described in Diamond are still useful even in a well indexed system such as we describe, but do not fill the same niche as a ranking algorithm, since they do not improve search precision.

LiFS [22] focuses on the problem of creating links between files, much as we do. However, in LiFS, unlike TaPIR, links are created either manually, by the user, or are driven by file content, requiring computationally expensive transducers to be run over every file, where we focus on links using data flow. LiFS considers the possibility of ranking using links, but does not provide a mechanism for doing so, and focuses on the applicability of application specific link types, much like Chirita [34] and Bhagwat [27] do in their desktop search research.

Finally, there are the semantic file systems [43, 44, 78], which present directory names based on the metadata of files, allowing the user to navigate and select files using their metadata. The original Semantic File System [43] used simple B-tree indexes to track attributes. The Hierarchy and Content file system (HAC) [**?**] was built on top of Glimpse [70], a system which performs two level text indexing with non-exact matches. The Logic File System [78] is built on top of Berkeley DB with B-trees. These are generally focused on user interface, rather than the performance of the index structures.

2.2.3 Other indexing

Indexing scientific data shares many challenges with databases as well as file systems. Column stores such as C-store [92], Vertica [65], HBase [12], or Cassandra [64], are one popular approach to dealing with sparse data. These have some advantages for scientific data, since they are well organized for tasks such as computing maximums, minimums, and averages, queries that scientists have indicated they are interested in.

WideTable [35] was specifically designed to meet the challenges of extremely sparse high-dimensional indexes, such as those required by scientific data. Patil *et al.* [83] also explored the question of appropriate architectures for searchable metadata in file systems. They suggested using BigTable [33] as the underlying storage for a file system. BigTable has good support for sparse indexes, and is highly scalable, thus might benefit scientific storage.

2.3 Provenance Collection

There are a number of systems currently in existence for collecting file provenance, primarily scientific in focus, such as Taverna [51], Kepler [20], and VisTrails [30]. There are also system level provenance collection mechanisms, such as transparent result caching [96] and PASS [73, 74]. These fall into two categories: disclosed provenance, where the user or application describes the provenance of a file, either before or after creation [20, 30, 51], and observed provenance [73, 74, 96], where the system is responsible for collection. Provenance can be collected automatically, or manually added, and provenance can be collected at a wide variety of semantic levels. In contrast to these systems, we are interested in automatic retrospective provenance, creating chains of provenance after the fact in order to study file system data flows. Our work is designed to be useful to researchers who wish to study file system provenance, but do not have the ability to set up provenance collection.

2.4 Ranking and Access Prediction

Our work in file system access prediction and indexing scientific data draws from work in desktop search and ranking, from ranking on the web, and from work on file and block access prediction.

2.4.1 Desktop ranking

The problem of effective ranking for desktop search has been a subject of interest for some time now. Researchers have identified the lack of connections between files, and proposed to infer links, based on file contents, directory structures, or browser caches [27, 34]. While these do add additional context for search, these approaches are computing intensive, as every file's contents must be parsed. They require manually encoding types of file relationships, such as that of header files to source files. They work poorly for binary data and proprietary formats, requiring a different parser for each type of file. Perhaps for these reasons, these techniques have not seen adoption, or even much in the way of evaluation.

Commonly used search tools, such as Spotlight [23], often rank based on the access time of the file, putting most recently accessed files first. Cohen *et al.* considered learning to rank, using combinations of basic features such as file names and access dates [36]. Google filed a patent on using recent accesses, combined with a weighting against access location [85], but did not take access frequency into account. While this is a convenient and generic approach, it does not take into account the frequency of usage. In this model, one access today is worth more than twenty yesterday. By contrast, we focus on both the age and the frequency of accesses, while still accounting for the idea of "working sets" and the importance of recent files.

More recently, the idea of tracking user activity as a way of improving search has gained some traction. Gaugaz *et al.* proposed a technique to temporally correlate file accesses [41], which is similar to provenance, but not as precise. Soules proposed the idea of using activity tracking to suggest additional search results by correlating files [90]. Dumais *et al.* suggested using visual user activity cues such as the file's time and author [39] to help the user quickly identify relevant search results. Lifestreams [40] suggests presenting a single time organized stream of data to provide visual and temporal context to the user. And Shah describes using provenance to aid in file system search [88]. All of these utilize user activity to improve the *recall* of queries, by drawing in additional search results. By contrast, our ranked search algorithms assume that the user has entered the right query, and that search recall is sufficient to find all relevant files. Instead we focus on improving the *precision* of the presented results.

Previous work has been done on provenance ranking, ProvRank and SubRank [71]. These two algorithms are also designed to determine importance within a provenance graph, and use the stationary distribution of a Markov chain, similar to TaPIR. However, these algorithms both focus on determining ranking in the context of provenance queries. Both are designed to prune provenance in the face of underspecified provenance queries. ProvRank and SubRank determine the rank of a provenance subgraph, using it to prune ubiquitous ancestor trees from the graph (such as the kernel compilation sub-tree, on which every file must depend), whereas we are interested in important files of all ages, and using these to predict accesses. ProvRank's approach is complementary to our approach, and in fact, can be used in conjunction with it. Pruning the provenance tree of extremely old files and processes can reduce the computational cost of TaPIR, in conjunction with rank smoothing to guarantee that every node has a value.

2.4.2 Ranking for the web

Ranking for the web has moved far beyond the days of Okapi BM25 [54, 55] and other term frequency/inverse document frequency similarity metrics, also known as *tf-idf* metrics. These types of ranking algorithms focus entirely on content, and ignore user activity, and relationships between documents. PageRank [79] is commonly considered the gold standard for general purpose web ranking. PageRank relies on the link structure of the web, much as TaPIR relies on the structure of a provenance graph. However, it makes assumptions about the structure of the link graph which are not appropriate in a provenance context, which has a strict direction of causality. In addition, because the search engine does not have direct insight into the usage of individual websites, it has to rely on links as a surrogate measure of importance. Since we have direct knowledge of which users access what, we can take advantage of the more detailed information.

Recent ranking algorithms have looked at online learning for ranking [18, 31, 53], relying on metrics such as click through data to determine, over time, whether the results presented are helpful and well ranked. In a web context, click through data is a surrogate statistic for usage data. These techniques can improve on an existing ranking scheme and might also be useful in a file system context, where click through data can offer feedback on ranking over time, but do not replace the need for an initial ranking of search results.

2.4.3 Access prediction

There are similarities of function between ranking and file-level predictive caching. Both predict what people are likely to access, but predictive caching focuses on current file system activity as a context for prediction, whereas ranked search uses the query as context, and maintains an overall list of predictions. Algorithms for ranking and caching may be able to complement one another, allowing the system to arrive at a better picture of the user's current activity and needs in order to pre-fetch files and choose relevant search results.

Algorithms for predictive caching often rely on a successor model [21, 46, 62], in which the most recent files are used to pre-fetch files which were previously opened after the currently active file. Like our work, these consider temporal relationships between files. However, we focus specifically on data flows and processes. In addition, these often use a fixed size cache, whereas we maintain a rank for every file, and allow files which were extremely popular in the past to maintain a small positive rank indefinitely. Yeh's work on using program and user information for access prediction [108] is similar in spirit to provenance. However, his work does not consider transitive importance, such as provenance based ranking can provide.

2.5 Naming and disambiguating

We are the first to propose managing file names automatically via the file system using metadata. Historically, the focus has been on using metadata to generate directory names, and file names have been left up to users and applications. By contrast, we focus on file names, and offer a generalized framework for both users and applications alike. The most similar areas of research to ours are those of application-generated names, web search snippets, and non-hierarchical file systems, each of which we discuss.

2.5.1 Application-generated names

Many applications such as iTunes [24], iPhoto [5], and Mendeley [7] currently generate file names, either for their own use or that of the user. However, none of them offers a generalized framework for file naming, and do not reflect outside changes to metadata. By contrast, TrueNames offers automatic naming as a service which any application can use, simplifying application development, keeping names synchronized with metadata regardless of source, and encouraging developers to export structured metadata.

2.5.2 Naming on the web

File naming can be thought of as analogous to the problem of disambiguating search results on the web. Web search, like file system search, has a huge number of files, (many with the same name, such as `index.html`), and when returning results the search engine must help the user choose between them. On the web, the historical assumption is that the user is retrieving textual information, and the common approach is to reveal a snippet from the page containing the search terms in context. Newer search types, such as video and audio, rely on a title and a key frame or excerpt. However, in the file system arena, these approaches are not feasible. Many files are in opaque data formats, and no snippet or key frame is available. Additionally, command line interfaces such as are common in file systems do not lend themselves to key frames or audio previews. Thus, we rely on file type-specific metadata to help the user identify their files.

2.5.3 Non-hierarchical and semantic file systems

The problem of naming directories has been a subject of interest for some time. In particular, semantic file systems [22, 43, 44, 78, 107] present directory names based on the metadata of files, allowing the user to navigate and select files using their metadata. For instance, the original Semantic File System (SFS) [43] treats all directory names as queries. If the user enters a query containing an unbound field name (such as `user:`), SFS will return `/jones, /root, /smith` and so on as subdirectories, automatically generating names based on lists of attributes and values.

Similarly, The Logic File System (LISFS) [78] establishes a *taxonomy* of attributes, such that some attributes subsume others. If the results to a query contain one or more attributes which are subsumed, only the higher level attribute will be displayed as a directory name, and only attributes which distinguish between the query results are shown. This is more similar to our disambiguation method. However, we rely on the template for a name, and add metadata only when required by a name collision.

However, none of these focus on the file names themselves, instead focusing on naming directories as a way to create queries. Our approach is designed to complement non-hierarchical systems, allowing files to be easily recognized regardless of context, and allowing non-hierarchical systems to disambiguate file names if needed, by adding additional metadata.

The most similar work is to what we propose is that of Jones *et al.* [57], who proposed a non-hierarchical HPC file system with automatically generated file names, chosen by examining the distribution of metadata fields. By contrast, our work uses a more robust and less complex scheme which puts the user and application in control of which metadata is used, and allows them to select attributes which are most appropriate for the file's semantic type, rather than relying on statistical techniques.

2.5.4 Faceted Search

Several of the semantic file systems just mentioned have been applying techniques from faceted search, a navigation model often found on the web for navigating databases and other well-structured data sets. ViewFS [61] explicitly focuses on file system search as a faceted search problem. ViewFS is also a semantic file system, offering virtual directories as query results. They propose using user feedback to refine the order of facet presentation over time. Research on faceted search such as Zhang and Zhang [109] and van Zwol *et al.* [98] has focused on learning facet ordering, based on relevance feedback via click-through rates. While these techniques work well on the web, where large number of users are expected to visit a site, they require sufficient user feedback for learning, and perform poorly when little or no data is yet available. Van Zwol does not offer a way to bootstrap initial facet presentation. Zhang and Zhang suggest using overall facet/value term frequency in the top n ranked documents, normalized by inverse document frequency, as a method of cold starting the facet selection. By contrast, we allow users and applications to manually select metadata attributes. It is possible that over time, TrueNames would provide enough data to be able to infer facets in a similar fashion, a question we leave to future work.

2.6 Summary

While there are a number of studies of file management behavior, none of them have previously focused on scientists and scientific data in large file systems, which we find to be different from typical desktop users. Scientific metadata, in turn, has characteristics which are distinct from either file system metadata or full text search, both of which are well studied cases. By doing a principled exploration of scientific metadata, we provide guidance both to our own research, and that of other researchers who wish to design search systems for scientific data.

Provenance is extremely diverse, and there are a large number of ways to collect it, manage it, and describe it. However, we are the first to create probabilistic provenance at the file system level, and to show how existing system tracing technologies can be used to create provenance records retrospectively.

Large scale ranking for file systems has not previously been explored. We have described a heretofore undiscussed connection between ranking and caching strategies, and are the first to use provenance data for either caching or ranking.

Finally, we are the first to describe a semantically coherent way of managing file names using metadata, which builds on earlier ideas in semantic file systems.

Chapter 3

Defining the Problem Empirically Through User Studies

The Harvard Law:
Under the most rigorously controlled conditions of
pressure, temperature, volume, humidity, and other
variables, the organism will do as it damn well pleases.

Anonymous

Governments world-wide spend hundreds of millions of dollars on supercomputers, each of them trying to outdo the other in speed. While speed is essential, it is not the full story. A supercomputer's utility must also be measured by its contributions to science and scientific discovery, which is in turn affected by the ability of scientists to use it efficiently and to its fullest extent.

We conducted interviews at a US National Laboratory, Los Alamos National Laboratories (LANL), as well as meeting with a scientist who works at the National Metacenter for High Performance Computing (NOTUR) in Norway. In total, we interviewed 8 scientists, and 7 scientific developers. Our aim was to uncover how they used supercomputing facilities, how they currently managed their files and metadata, and what their points of frustration were with existing scientific file systems. Interviews were semi-structured, with a series of prompt questions, as well as free-form discussion. In particular, we asked them about directory structures, about data sharing practices, about what types of metadata they collected, and where they stored it, as well as how they used it. Finally, we asked them what they wanted to change about file systems.

We find that scientists and scientific developers are frustrated with the current state of supercomputing file systems in a number of areas, and that scientists form an invisible and weak link in scientific data management. The users we spoke with described problems with data sharing, with metadata management and robustness, and search, each of which took time away from their primary task, science. In addition, many of these problems required the user to use brute force tools such as `find`, `du`, and `grep`, adversely impacting file system performance. Based on our interviews, we find that by adding new features to scientific file systems, we can ease the lives of scientists and allow them to be more productive, while reducing burdens on the file system.

3.1 Background

High performance computing is primarily computation of simulations. The simulation is divided up into small pieces, with each CPU responsible for a piece. Computations are tightly coupled, requiring constant intercommunication between CPUs. Given the large volume of hardware involved, failures occur frequently, requiring computations to be *checkpointed*, writing enough information to the file system to be able to restart an in-progress computation from a known state.

Both LANL and NOTUR have a number of special purpose file systems, intended to be used in different ways. LANL uses a Network File System (NFS) [86] based on NetApp filers [48] for smaller files that are written sequentially. The NFS file system contains home directories and project directories. The Panasas Parallel File System (PFS) [102] is a high performance parallel file system shared among multiple supercomputers, and is meant for large files and parallel writes, such as check points. It is considered a scratch file space, and files there are meant to be archived or moved to NFS after a short period of time. Additionally, LANL has archival storage space, using HPSS [101] and GPFS [87].

NOTUR, like LANL, has several different supercomputers, with a shared parallel file system, meant for scratch space, and home directories on NFS, as well as an archive.

The file systems have a stated usage policy, which is for large files that are written in parallel to be kept on PFS, and smaller sequentially written files to be stored on NFS. However, there's a great deal of variance in actual usage. Some users simply ignore the policies, and store everything on the parallel file system. Others try to follow policies, but with mixed results. As we will see in the sections on sharing and search, having multiple active file systems causes issues for users.

3.2 Data Sharing

Many users (73%) specifically called out shared data as a problem. It was challenging to set up sharing within a group, or for new users to figure out how to share their files with their mentors for assistance. Several users were frustrated by limitations on the number of groups they could create in order to share files. Having multiple file systems required users to set up sharing individually on each system. Most users shared their data with someone else at least some of the time, and several worked in large collaborative groups. Many of these problems could be solved with current tools such as ACLs, but there are still significant usability concerns around setting up sharing, selecting files to share, and creating semantically coherent sharing across multiple file systems.

We also asked users how easy it would be for someone else to take over their work if they were to suddenly become unavailable. Many users were nonplussed by this question. (One of them looked quite concerned and said he'd never thought about it.) Users who worked in large groups fared better, saying that someone else in the group could bring a replacement scientist up to speed. However, a significant percentage of our respondents (20%) said it would be challenging or impossible for another scientist to take over their work without having been given a guide to their metadata and file organization system. As we describe in the next section, most users manage their metadata and files in an ad hoc fashion, outside the file system.

Based on our discussions, existing tools do not make it simple or intuitive to share data or metadata, especially across multiple file systems.

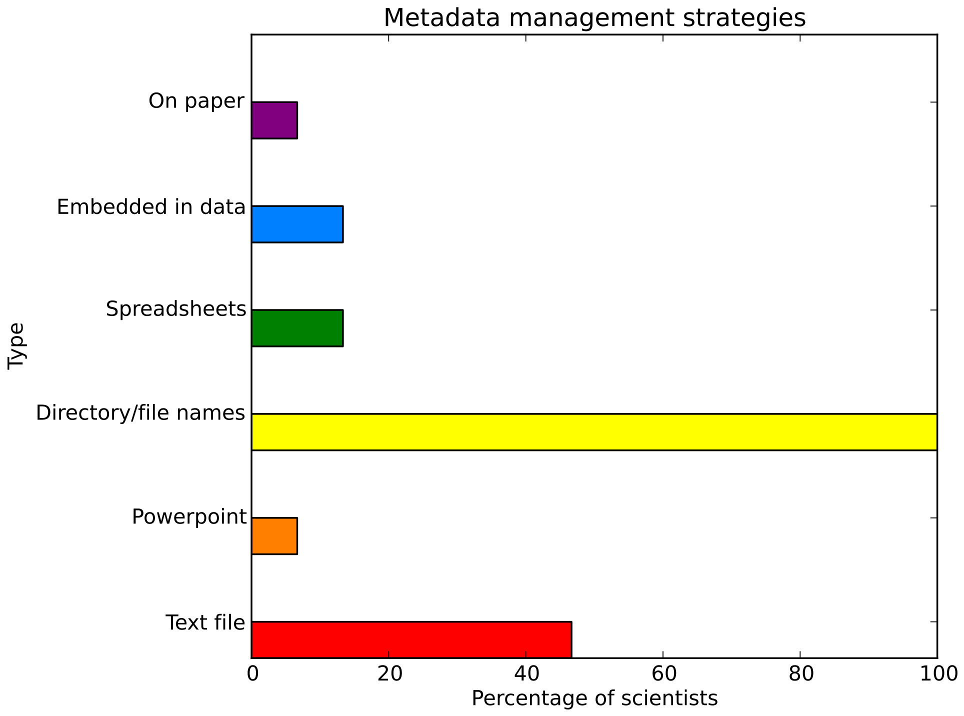


Figure 3.1: Metadata and file management strategies used by scientists. A scientist may use multiple strategies, so percentages do not sum to 1.

3.3 Metadata and File Organization Methods

The majority of users we spoke to (66%) maintained some sort of external tracking system for managing their files and associated metadata. Metadata was extremely diverse, including structured information such as parameters, provenance information, and multimedia information such as stills or video clips.

Metadata management methods ranged from text files, stored alongside the data, to spreadsheets, Powerpoint, and 3-ring binders, and were stored on NFS directories, personal computers, and bookshelves, as summarized in Figure 3.1. In addition, 100% of users embedded additional metadata into file names and directories. In some cases, this metadata was not available anywhere but the file name.

Embedding metadata into directory names and files allowed users to find their files, and share metadata with others, but also meant that metadata was often manually created and added, with occasional errors that might have been avoided if the metadata were created directly from data or programs. Another concern that users voiced was that of manually duplicating directory structures between file systems, as shown in Figure 3.2. For instance, users said that sometimes they accidentally inverted or omitted the nesting of metadata, resulting in problems when moving files between file systems.

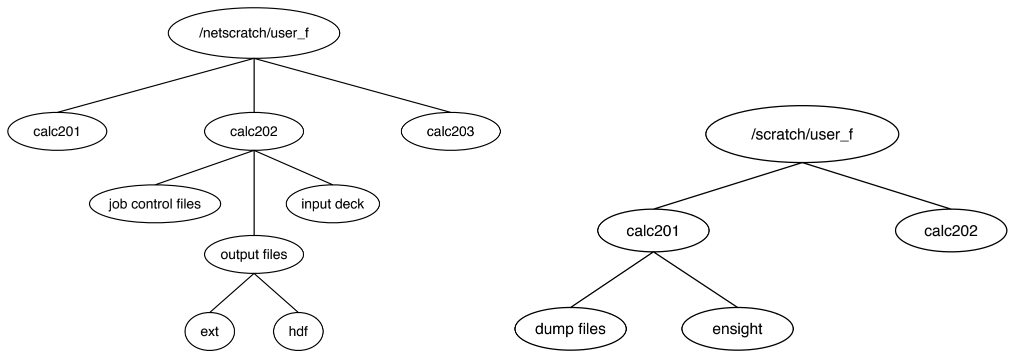


Figure 3.2: Parallel directory structures

These metadata management strategies highlight several problems. Firstly, the file system does not provide good methods for managing file metadata or collecting metadata automatically, forcing the users to create *ad hoc* methods for storing and managing metadata, and then spend large amounts of time curating these metadata stores. These methods are often undocumented, and particular to a certain user or group. Additionally, current metadata management is *brittle*, *i.e.,* there is a high probability of metadata loss, especially for the 33% of users storing some or all of their metadata in paper notebooks or on their personal computer. In most cases, metadata can be reconstructed from the data, but at a high cost, and in some cases, it may be lost forever. Finally, users prefer to organize and navigate their files by metadata, but need better tools for doing so. In summary, existing tools drive users to store metadata in ways that are hard to share, hard to interpret, and brittle.

3.4 Search

Nearly all of the users we spoke with were frustrated with their ability to find files (93%). The remainder had created their own search tools, specific to their needs,

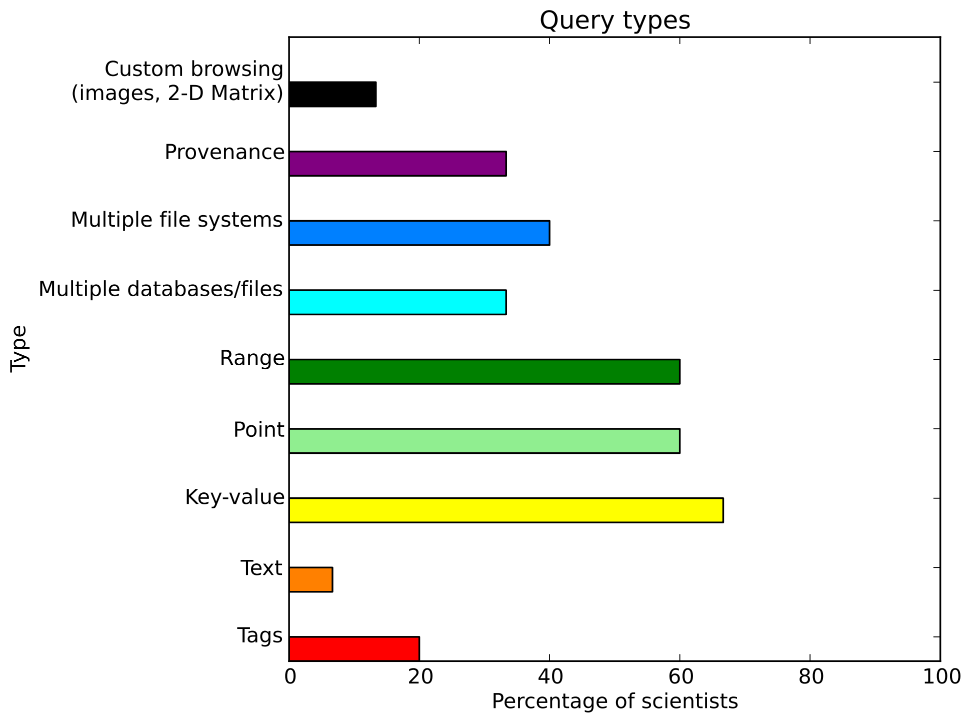


Figure 3.3: Types of queries scientists are interested in performing. Scientists may be interested in multiple query types, so percentages do not sum to 1.

which filled the gap left by the file system. Scientists had a variety of tools for finding data within a single file, but few of them were satisfied with their ability to find the files themselves. Cited issues included problems such as fragmentation across file systems, the inability to do things like range queries, the desire to do provenance queries on a wide range of file types, and the inability to associate multimedia with files in order to do multi-modal search.

While scientists are typically looking for data they already know exists, they say it is still challenging to find. They need to start by remembering if the data is in primary, parallel, or archival storage, or commit to searching each of them independently. Then they must attempt to remember something about the file name or directory, or some piece of POSIX metadata which can help them identify it. In the worst case, they may have to resort to manual search, opening individual files to determine the contents. Many scientists prefer not to rely on search or navigation at all, instead using an external metadata tracking system, such as those described above, to manage their most important files.

Archive search was a particular problem for most users. Since archived files may be on tape, retrieving files from archival was slow. It was prohibitively expensive to apply brute force searches such as `grep`. Many users stored their archival files in TAR files, meaning that the TAR had to be retrieved and its table of contents had to be inspected to look for a given file. In addition, the archival file system had different constraints on file sizes, meaning that files might have to be renamed on migration, which might result in metadata being removed from the name or being lost. Users went to great lengths to avoid having their data moved into archive, in order to avoid having to find it again [49].

We summarize statistics on query types users wanted made available in Figure 3.3. It is noteworthy that very few scientists expressed interest in full text search. We observed two potential reasons for this. One is that scientists, more so than most users, work primarily with opaque binary formats, making free text search less useful. Another is that available tools for text search, while not ideal, are at least sufficient for many users, so it is not perceived as a pain point. By contrast, there are very few tools available for structured search, and especially for provenance search or multimedia browsing.

Giving users tools to do structured search over metadata and provenance can significantly improve their effectiveness. In addition, federated search tools across multiple file systems are essential. Archival file systems in particular can benefit from fast indexed search. Many of the necessary technologies are already available, but either have not been a priority for HPC file systems, or have scalability concerns that must be addressed.

3.5 Conclusions

In light of these interviews, it is clear that better file and metadata search and management capabilities are needed for scientific file systems. Lack of support for file and metadata management has resulted in a variety of ad hoc manual processes, which are time consuming and error prone, and a point of frustration with scientists. Existing file systems make it difficult to share data, store and search metadata, and manage files across multiple file systems.

Better support for file management not only improves matters for scientists, it can reduce demands on the file system. By reducing reliance on slow brute force tools such as `grep` and `find` on primary file systems, and making it easy to find archived files without retrieving data from tape, the file system can reduce unpredictable loads from user queries.

Integrating metadata storage and indexing into the file system allows users to tightly couple metadata and data, and automate what is currently a primarily manual task. Allowing users to easily transfer metadata into file names and directories, using non-hierarchical file systems and automatic file naming, will improve organization and reduce errors. By creating a unified search space, scientists can always find their files, regardless of where they are stored. Scientists need access to much richer query interfaces, allowing them to mix keywords, provenance, and key-value searches for points and ranges. A richer file browsing interface, one which allows them to associate and view images and video clips, will also allow them to manage their files more effectively.

By making rich scientific metadata searchable and easy to store, we can assist users in managing their metadata inside the file system, rather than outside it. This can make it easier to find files, and share files and metadata. It reduces the human cost of metadata management, and the risks of external metadata management. Finally, it reduces the load on the file system itself, allowing scientists and systems alike to focus on what matters: science.

Chapter 4

Characterizing Scientific Metadata

O! be some other name:
What's in a name? that which we call a rose
By any other name would smell as sweet

William Shakespeare

We examined a wide range of metadata studies, and found that none of them focused on rich metadata, and particularly on scientific metadata. There were some high performance search systems based on these studies [50, 67, 75, 97], but they used POSIX metadata, and then attempted to extrapolate performance for other use cases. Thus, we have studied scientific metadata directly, in order to gain a more realistic understanding of the design space of scientific metadata and content indexing systems. Having studied scientific data, we consider existing indexing solutions for both file system search and scientific search, and conclude that they are not well suited for HPC search.

Consider scientists working together on a shared HPC computing and storage system. Scientists may want to search files by size or age, but are more interested in searching metadata about content, as we discuss in the previous chapter and other work [93]. An astrophysicist might search for files with a certain peak brightness. An oceanographer might want files containing observations at a certain salinity, and a biologist might need files about a species or watershed area. Each of these searches is a metadata search, but rather than relying on system generated metadata, it relies on metadata that is domain specific, and may be embedded in content or stored in an extended attribute.

We find that in contrast to POSIX metadata, scientific metadata is heterogenous, with different metadata for different disciplines. It is sparse, with many missing values even within the set of files from a single discipline. It is high-dimensional, and those dimensions are a mix of numeric, textual, and categorical data. It can also be non-atomic, with many values for a single field. We also find that it has a number of features that an indexing strategy should take advantage of, such as compressibility.

Search for scientific metadata poses a number of unique problems. Scientific metadata is large, and can easily outstrip the size of the data it describes. In many cases file content and metadata are indistinguishable. For instance, in astronomy a dataset may contain raw data such as pixel brightness, and metadata such as a catalog identifier. What matters is whether the data will be queried. We focus on search for data in *fields*, a single dimension of metadata such as temperature or author, as opposed to free text with a document. Scientists have expressed a strong interest in structured search over scientific data, whereas since free-text search at scale is a well explored problem.

File system search, especially at scale, poses additional constraints. Results to a file system query are in the form of a list of matching files, rather than a set of metadata fields. This is different from many common RDBMS database indexing problems, where results are records in normalized tables, and often require eventually performing a join to retrieve some or all of the fields. File system search does not require complex transactions, and the query model is relatively simple, since each file's metadata is independent. Finally, HPC file systems are often operating near full capacity, and cannot use a large fraction of storage to store indexes, or a large portion of disk and CPU bandwidth for brute force search.

A search system for scientific metadata must balance each of these constraints. It must make it possible to query a large body of sparse, high-dimensional data without consuming a large fraction of CPU, I/O bandwidth, or storage. It should be able to add new files to indexes efficiently. Finally, it must efficiently handle a mix of data types.

In addition to providing a new study of scientific data, we examine a variety of indexes, and conclude that column stores are the best match for both the characteristics of scientific data and the types of queries scientists have expressed interest in.

4.1 Experimental Design

Before going further, we provide background information about our experimental design. Our experiments explore a wide variety of data in many formats, and indexes which are designed with very different goals. We define some common terminology which can be used to describe a variety of data formats. We describe the index types we examine, as well as the data sets which we analyze, the databases we consider, and the tests we perform.

4.1.1 Terminology

Since different index structures organize data differently, it can be hard to find a common terminology to discuss them. In addition, metadata can refer to different things when used in a scientific context, rather than a file system context. We will discuss *fields*, a single dimension of metadata such as temperature or author. *Metadata* refers to data in fields, both extended metadata and metadata embedded in file contents. We refer to a single data object, typically a file and its metadata, as an *item*. And we use the term *data element* to refer to data in a single field of a single item. A field of an item may have multiple data elements, as we will discuss later.

4.1.2 Index Types

We consider the suitability of a number of index and database types for file system search and scientific indexing. We describe each of them here, along with a brief description of their strengths and weaknesses, summarized in Table 4.1.

Row stores, such as the relational databases most people are familiar with, store items in row-order on disk, and employ additional indexes for fast search on specific fields. Row stores are optimized for retrieving the entirety of every row that matches. They support complex transactions and query models. Inversion [76] was the first system to propose integrating indexes into the file system, and used PostgreSQL, a popular open-source object relational row store.

Column stores are a more recent database design where data is laid out on disk by column. They support statistical queries better, use less space for sparse data,

Data	Column stores	Row stores	Spatial trees	Inverted indexes	HDF5	FastBit
High dimensional	Yes	Yes	No	Yes	Yes	Yes
Sparse	Yes	Nulls	No	Yes	Yes	Nulls
Multiple values	Yes	Foreign key	No	No ranges	Yes	Yes
Non-numeric	Yes	Yes	No	Yes	Yes	No
Indexes						
Range queries	Yes	Yes	Yes	No	Yes	Yes
Specialized indexes	Yes	Yes	No	No	No	No
High compression	Yes	No	No	Yes	No	Yes

Table 4.1: Comparing index types

and can efficiently answer queries which use only a few terms, because only the columns present in the query need to be loaded. However, they do not offer good support for transactions and joins, and perform poorly when an entire row needs to be recomposed.

Spatial trees are designed for indexing multi-dimensional numeric data, such as geographical information or points in a numeric space. They are good for answering range and proximity queries on spatial data, and can be used for clustering. Spyglass [67] and Smartstore [50] were the first to suggest using spatial indexes for metadata.

Inverted indexes are designed for textual data, but can be used for any sort of point query. In an inverted index, each keyword or value has an index, containing a list of pointers to matching documents. In order to implement inverted indexes for structured data, each field must have a list of keywords and associated indexes. Inverted indexes are excellent for sparse data. Since the structure is simple, inverted indexes can also be implemented on top of other database structures. For instance, a relational database might have a table containing keywords as a primary key, and a column containing a list of documents. Columns in column stores are often implemented as a `value:rowid` inverted index.

FastBit [105] is designed explicitly for scientific databases. FastBit is a bitmap index optimized for high dimensional scientific data, which performs extremely well on numeric data, and is very compact. It performs well on point and range queries, but does not support non-numeric data.

HDF5 [17] is a popular data format for storage of array based scientific data. Recent versions also have the ability to create B-tree indexes over a specific field of the data in order to search within a file. While it is not an index structure *per se*, we will discuss its merits as a way of searching scientific data as well, since it is a popular solution to scientific data storage and search.

4.1.3 Data Sets

	Discipline	Native format	Record count	Sampled	Sample count	Total size	Fields
Dryad	Biology	XML	31k	No	31k	400MB	44
WISE	Astronomy	CSV	564M	Yes	10k	1TB	285
ARGO	Oceanography	NetCDF	2B	Yes	634,880	330GB	108
ORNL	Climatology	CSV	1478	No	1478	154kB	14

Table 4.2: Data set characteristics

In order to get a representative sample of scientific metadata, we chose files from Dryad [11], the Wide-field Infrared Survey Explorer (WISE) All-Sky Release [14], Argo [15], and Carbon-14 observations from Oak Ridge National Laboratories [45]. Dryad was further divided into data using The Open Archives Initiative (*OAI*) Protocol for Metadata Harvesting [9], and the Metadata Encoding & Transmission Standard (*METS*) [13]. We chose these data sets because they are readily available, and cover a wide range of science one might find in a large computing installation. Our goal is to characterize what could be expected in a system where one or more of these data types is resident. While some scientific computing installations may handle only a single kind of data, the largest installations support a wide range of scientific research, and will have indexes which must support multiple types of data.

While our choices were made on the basis of breadth, we also find that they each have very different metadata characteristics, offering a broad perspective on index requirements. We summarize their characteristics in Table 4.2. Where we subsampled, we did so in a uniform random fashion. We make the simplifying assumption that the metadata is uniform with regard to the characteristics we are studying, since our chosen characteristics are largely the product of scientific practices in a given field.

To analyze the data, we decomposed each data set into a columnar format by generating sorted lists from each field. Where data sets were natively in a non-tabular format, such as XML or NetCDF, we generated a list of all available fields, and then grouped elements into sorted lists based on tag or array name, respectively. For row analyses, we used the top level document item and the array position, respectively, to create a row ID.

4.1.4 Sparsity

Many indexing systems assume a fixed schema, where all fields are present for all items of the same type. We examine the *sparsity* of fields, how often values are missing from a given field. To calculate sparsity, we determine the total number of fields possible for any item of that type, and then calculate what number of those fields were actually present in each of the individual items of that type.

Sparse data is challenging for spatial tree based indexing schemes, which cannot place data with missing values in the tree, and must use estimation to fill in missing values, creating fake data that must then be stored [94]. A single table row-based index will also have space implications. Even if a table is built for each data type, it will still need to store nulls for missing values [35]. A normalized schema can represent this data more efficiently, but at the cost of complexity. Although bitmap indexes are capable of storing data using masks, such that an element is only stored when data is present for a field, FastBit does not currently have mask support [16], and must store a null value. Inverted indexes and column stores only store data when data is present for that field, and will have space savings for sparse data.

4.1.5 Atomicity

Atomicity describes how often a field can be present for an data item (*i.e.,* a file), and whether it can be represented as a single point. An *atomic* value is one where only zero or one point elements can be present in a field. For instance, tabular data is strictly atomic, since it can only support one value per column in a row. Non-atomic data may represent an list of many elements in a field, such as a list of authors, or a single range element with boundaries, such as a geographic area, an uncertainty range, or a start and end date.

Different indexes handle list and range values in a variety of ways. Some cannot represent them at all, while others require careful schema design to represent and search them efficiently, such as normalization to represent lists. Non-atomic data requires an index design that can represent multiple values or ranges, and match them to queries. Thus we analyze the *atomicity* of fields, *i.e.* whether a field can be present multiple times in a single item, or represent a range.

4.1.6 Entropy

Entropy serves two purposes. First, it allows us to consider the selectiveness of queries. Fields with low entropy do a poor job of reducing the results size, whereas high entropy fields can be very selective. Secondly, it allows us to characterize compressibility. Low entropy data is easy to compress, and is less expensive to store. Entropy is agnostic to data type, which allows us to do direct comparisons of string and numeric data.

We chose to focus the entropy of whole values, to study query selectiveness. Whole value entropy does not entirely capture compressibility, but serves as an upper bound. More compression is often possible for fields which can be segmented or bucketed, such as free text and numerical data. We examine block-based column and row entropy, such as a database might use during compression, to compare their effectiveness. We discounted null values, since these are not relevant to selectivity.

We examined the entropy within each data set in two ways, using the bit-wise Shannon entropy

$$H(x) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$$

where $p(x_i)$ is the probability of value x_i occurring. For columns, we sorted the values for each field (corresponding to a column), divided it into 4kB blocks and calculated each block's entropy. For row entropy, we took all fields present in an item (excluding nulls), and took enough rows to fill a 4kB block, then calculated entropy.

This allows us to consider the effects of column organization versus row organization for compression. Field entropy also characterizes the selectiveness of a query on that field. Fields with very low entropy are very non-specific. In the worst case, an entropy of zero, where every value is the same, that field does not allow us to eliminate any possible results. By contrast, very high entropy fields, such as a unique identifier, quickly narrow the search space. In between are fields which are moderately selective.

4.1.7 Data Types

Finally, we look at the overall distribution of data types for fields, both in terms of storage and semantic data types, to determine necessary index types. We examine the data types of the fields, based on the data set documentation. Knowing the distribution of field types is useful for index choices. Different index designs are better for text versus numeric data, and there are efficient specialized indexes for data which is geospatial or time based. To examine type distributions, we categorize fields as numeric or string data. We then further examine them to determine if they are geospatial, a set of flags encoded as a string or number, *etc.*

4.2 Analysis and Results

Our tests left us with some clear winners and losers. Scientific data is extremely sparse. Scientific data can have fields with a large number of values per item, such as lists of authors and species, and some data sets contain range values. Scientific data has a wide range of entropy, but has many low-entropy fields that can benefit from compression even after accounting for sparsity. And finally, scientific data has a mix of data types, including text, numbers, geospatial data and binary flags, which necessitate indexes which can handle a wide range of types.

4.2.1 Sparsity

We find that even within a single discipline elements tend to be sparse, as shown in Figure 4.1. Only a few fields are present in all items, and these fields tend to be unique identifiers for the system's use. A poor choice of index will waste space storing nulls, or be unable to represent the data at all. This allows us to rule out indexes such as kd-trees, which cannot represent sparse data, and supports the use of inverted indexes and column stores.

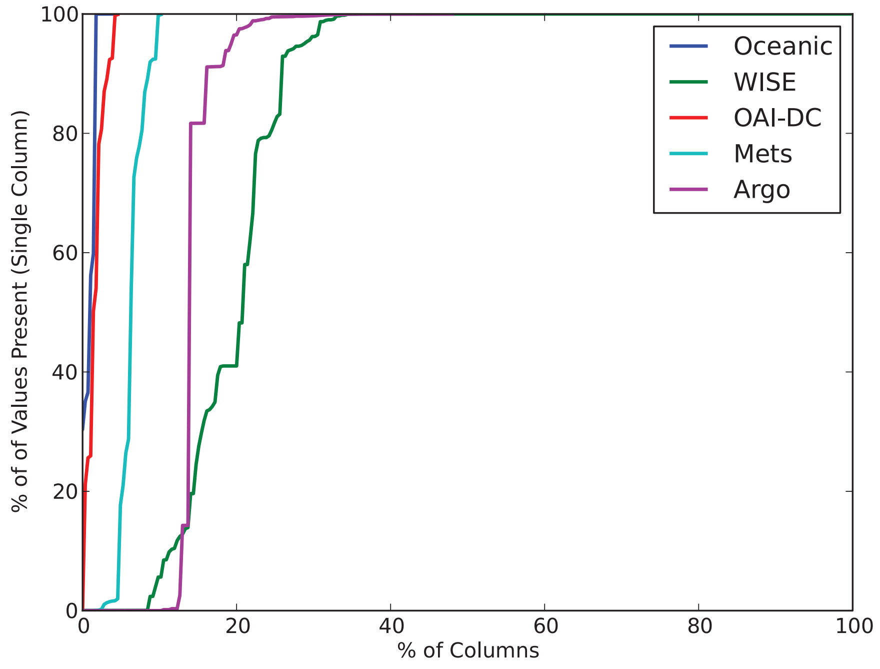


Figure 4.1: Sparsity for all data sets. For a randomly chosen element from $X\%$ of columns, there is a $Y\%$ chance it will not be null.

4.2.2 Atomicity

While most of the data sets we examined are strictly atomic, the biology data have many fields with multiple elements per field for a single item, as shown in Figure 4.2. Only 55% of Dryad fields are atomic, *i.e.,* have a single entry. Less than 1% of fields have more than fifteen entries, but can go much higher. Finally, one of the numerical fields in the ORNL data mixes ranges and point values. The distribution of non-atomic items is correlated with the field. Some fields are commonly non-atomic, while others are commonly a single point value.

Any system which supports a variety of scientific metadata must handle non-atomic data. Both list entries and range entries will cause problems for spatial trees, which expect point values. At best, a spatial tree will require an additional data point for each value, leading to duplicated data and consistency problems. Inverted indexes

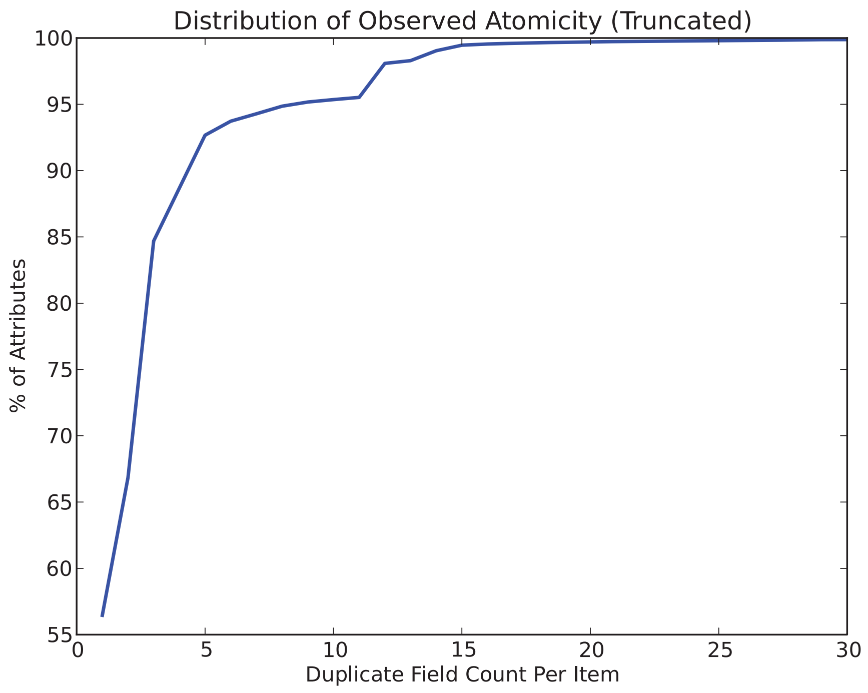


Figure 4.2: Field value counts for both OAI and METS data, truncated on both axes to show detail, since one item in our data has over eight hundred species associated with it. Indexes which cannot handle multiple values for a field for a single entry will be unable to store this data, and a poor choice of index design can result in large amounts of duplicated data.

cannot represent range values at all. With properly normalized schema design, an RDBMS can support list and range values. Column stores and bitmap indexes will also need schema design to handle range values, but can support high cardinality list values in a single column given a simple schema, and are a better choice.

4.2.3 Entropy

As Figure 4.3 shows, row entropy is very low variance. This is intuitive, since taking entropy over all fields will average the entropy of those fields, leading to lower inter-block variance. However, mixing different types increases total entropy, meaning row data will be harder to compress. Column stores, inverted indexes, and bitmap

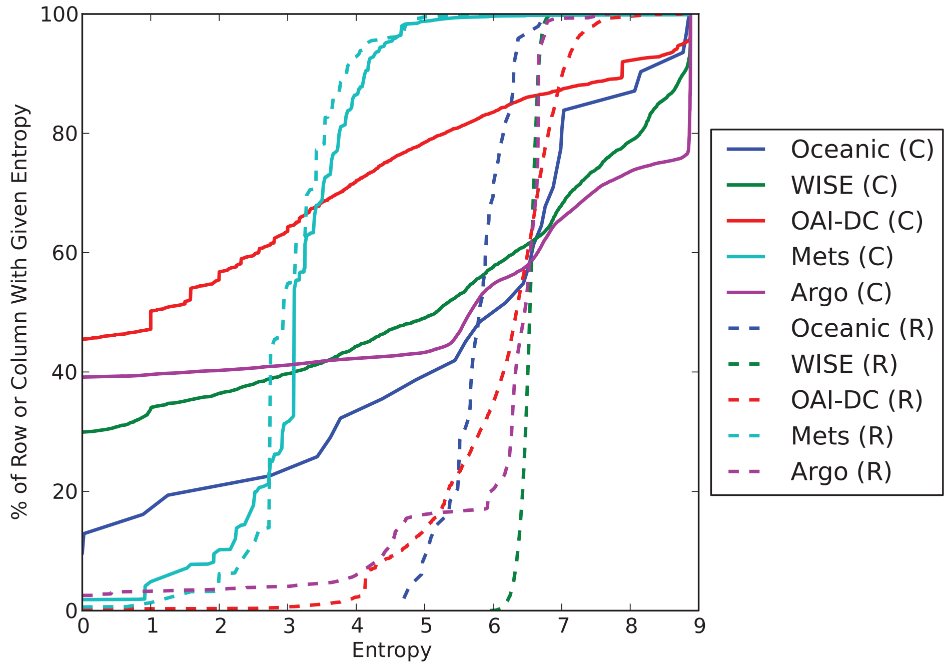


Figure 4.3: CDF of entropy for all fields in each data set. Dashed lines are row entropy, and solid lines are field entropy. Fields consistently have lower entropy than rows, indicating they will compress better, allowing more indexes to be stored in the same amount of memory or disk. However, field entropy varies significantly, suggesting that some fields are much more useful to query (and therefore index) than others.

indexes can take advantage of the structure of low entropy data in order to compress it. Some spatial trees can take advantage of data entropy to balance trees efficiently, but do not compress the data itself, and row stores cannot take full advantage of low entropy columns, since they must average entropy over all columns. Some types of compression can compress data on a per-field basis, while allowing data to be stored in row order, but will require more accesses to decompress data.

4.2.4 Data Types

Scientific data contains a mix of data types, including geospatial data and binary flags, which necessitate indexes which can handle a wide range of types. In

		Dryad	WISE	Argo	ORNL	Totals
	Fields	46	285	108	14	453
Storage type	String	100%	4%	62%	29%	28%
	Numeric	0%	96%	38%	71%	72%
Semantic type	Date	2%	4%	7%	7%	5%
	Spatial	2%	9%	2%	21%	7%
	Flagsets	0%	19%	14%	0%	15%
	Native types	96%	68%	77%	72%	73%

Table 4.3: Data types in scientific data for all data sets. We examine storage types, and semantic types that can have specialized indexes. String and boolean support is a must, and having native index support for time and space can significantly speed up queries.

Table 4.3, we show the distributions of raw data types and semantic types.

Indexes which cannot handle string data, such as bitmap indexes and spatial trees, will not suffice for indexing scientific data. On the other hand, spatial indexes can be very useful as part of an indexing system, and bitmap indexes can be very efficient for flag sets and numeric data. The ideal system will take advantage of the semantics and structure of each field to search efficiently over heterogeneous data.

4.3 Conclusions

Based on our findings, we conclude that column stores are an excellent fit for scientific data, and can answer file system search queries in a very efficient fashion. Scientific metadata is large, sparse, heterogenous, high entropy, and high dimensional, making it significantly different from POSIX metadata. Existing approaches to file system indexing, such as spatial trees and row major databases, will perform poorly for indexing scientific metadata.

Scalable searchable file systems are crucial for scientific computing, and understanding the characteristics of data is essential in making good design decisions. Our findings will significantly benefit researchers in the area of scalable scientific indexing.

Chapter 5

Extracting Provenance from Existing Data

Hey Rocky, watch me pull a rabbit out of my hat!

Bullwinkle

Provenance is the set of data collected to answer the question "how did this object come to be here?" In the art world, it may describe a chain of ownership and transportation. In computing, especially scientific computing, it refers to questions about data flow, describing the objects and processes which went into creating a given piece of data, such as a file, database entry, or scientific result. Provenance may be at the level of workflows, describing individual data items and equations, at the file system level, describing files and system processes, or at a variety of other levels of detail. Provenance may also be *retrospective*, describing what has happened, or *prospective*, describing what should happen, such as creating a workflow to later be executed.

When it comes to provenance, especially at the file system level, there is a chicken and egg problem. System designers are loath to add provenance to systems, citing performance and space overheads, and a lack of useful applications. At the other end of the spectrum, researchers who might design useful applications and performance enhancements for file system level provenance find it difficult to justify their work, since there are very few system provenance workloads available for evaluation and analysis, and most available workloads are very short and and focused on specific types of scientific computing.

We propose to bridge the gap between provenance data and provenance research, by demonstrating how existing system traces can be converted into retrospective file system level provenance. There is a large extant body of file system traces used by researchers, which can be used to study system level provenance over longer periods, and adds diversity to the types of available workloads.

While provenance information, such as command line and environment parameters, is commonly collected by provenance systems such as the Provenance Aware Storage System (PASS) [74], that depth of information is not collected by most system traces, and is extremely expensive to store. Instead, we focus primarily on the subject of *data flow provenance*, tracking the flow of data between files and processes to create a provenance graph. Data flow provenance graphs can be useful for research in a variety of areas. Some current areas of research which use provenance graphs include the data flow based file ranking we study, as well as transient provenance [56], provenance graph compression [106], provenance pruning [71], and many other topics. By creating detailed data flow provenance for computer systems, covering longer periods of time, we can study currently unanswerable questions about repeating structures in provenance graphs, the evolution of provenance graphs over time, and so on.

We describe how an existing system call trace can be converted into a file system data flow provenance graph. We demonstrate our technique on the LASR system call traces [63], a set of system call traces collected over the course of a year on several mobile computers. We show that our technique achieves 92% coverage of process creation and 98% coverage of file opens. We compare the created provenance graph with directly collected provenance traces from PASS [10], and show that our provenance graph has similar depths and branching factors, meaning that our coverage factor does not significantly alter the characteristics of the graph. Converting the LASR traces creates the single largest set of publicly available provenance traces, covering a longer time span than any other data set, which makes them a invaluable resource for provenance researchers.

5.1 Architecture

Our conversion architecture consists of a trace parser, a state tracker for files and processes, and a provenance graph representation which is independent of trace format, as shown in Figure 5.1.

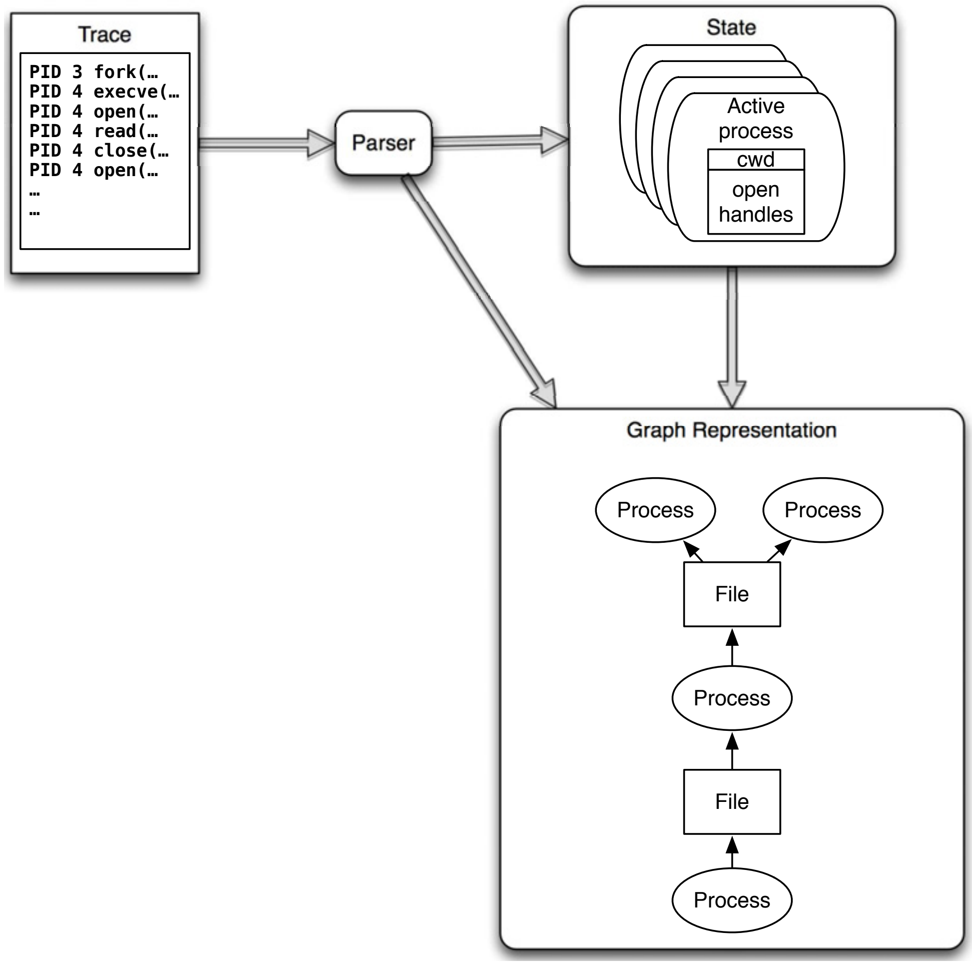


Figure 5.1: Architecture of our provenance conversion tool. Inheritance arrows flow from children to ancestors.

5.1.1 Inferring provenance

There are many different ways for data to flow between processes, as security researchers have long known. The presence or absence of a lock file is data, as is the contents of a command line argument, or even the timing of a packet. However, we focus on the broad structures of data flow: reads and writes, via files and pipes. We also consider forked processes to constitute a form of data flow. Inter-process communication via shared memory is also a data flow mechanism, as are network sockets, but since neither the LASR traces nor the comparison traces we studied trace any IPC system calls, and LASR did not trace network sockets, we have left handling of those calls for future work. We consider the set of system calls listed in Table 5.1 to constitute information about data flow.[1] We treat directories as special files, and track which processes created them, but do not treat directory reads or writes as data flow to the directory.

Table 5.1: File and Process System Calls

Processes	`fork, execve, exit`
Files	`creat, open, write, read, mknod, unlink,` `truncate, close, pipe, mkdir, rmdir`

However, this is not sufficient to create a full provenance graph. In order to infer provenance after the fact, we must be able to recreate as much of the file system state and I/O state of a given process as possible, including working directories, links and soft links, active file descriptors, and so on. Thus, we also track calls which alter the state of process I/O and the file system, listed in Table 5.2, in order to interpret arguments to the calls listed above. For instance, if a file descriptor is duplicated we annotate the new file descriptor with the file name used when opening the original descriptor. If a link is created to a file, we attach the link target's provenance to that name. We track working directories for processes, in order to resolve relative file references. This state tracking allows us to discover trees of data flow, from process to file to process, as well as uncovering as much data as possible about the state of processes and the file system.

In order to create provenance, we replay a system call trace. We generate a

[1]Where a system call has more than one variant, such as `pipe/pipe2` or `chdir/fchdir`, all variants are implied.

Table 5.2: State Altering System Calls

Process state	`chdir, dup`
File system state	`rename, symlink, link`

node in the graph for each file or process that we observe. This node stores a variety of information, such as a type (file or process), a first observed time, a last modification time, a unique identifier (a file name for a file, or a combination of process ID and first observation time for a process), and finally and most importantly, a list of directed *edges* which have either transferred data in or out of a node.

Edges in turn have a time, an origin and destination, and a tag indicating the type of operation they represent (create, delete, open, read, or write.) The first two can apply to file or process destinations, while the last three are reserved for edges whose destination is a file. One process may have multiple edges to the same file, representing different operations, or even the same operations at different times. (For instance, a process may open a file, read it, close it, reopen it, and read it again. This will result in five edges being created, two opens, two reads, and a close.) We do not do any detection of duplicate edges at this time.

In addition to storing a node, we also create temporary data structures to store the state of processes. These data structures allow us to map open file handles to file names, and to track the current working directory of the process in order to resolve relative paths.

5.1.2 Handling missing data

System call tracing is not always perfectly reliable, especially tracing using the `PTRACE` framework, and system calls are sometimes omitted from the trace, particularly near `fork` events. This means that not all state changes will be observed. This can take several forms. Keeping track of certain types of observable omissions can help us evaluate the accuracy of our conversions.

One possibility is that we may miss a process being created, or a file being opened. We can infer that this has occurred after the fact. For instance, if we observe an system call from a process ID which we have no previous information about, we can infer that there is a process with that ID, although we cannot tell which of the other processes forked it. If we see a process reading from a file handle that we have no information about, we can infer that a handle was opened, but cannot tell which file they are reading. Thus, we have processes and files which are *orphaned*. We track these orphaned nodes in order to acquire statistics about the accuracy of the traces, and thus our graph construction. In general, we assume that our missing data rate can be estimated based on the highest observed error rate for process creation and/or file opens.

5.2 Experimental design

We tested our algorithm using a series of traces known as the LASR traces [63], collected on several general purpose mobile computers over the course of approximately a year. The LASR traces are ideal for this purpose, since they contain not only information about processes and file opens, but also reads and writes. This makes them similar in graph detail to tracing systems such as the Provenance Aware Storage System (PASS) [74]. Although they do not contain information about the environment or command line arguments as PASS does, the LASR traces cover a much larger time span than the longest available traces from PASS.

As our point of comparison, we selected a series of traces created by PASS [10], described in Table 5.3. These cover a more clearly defined set of workloads than the LASR traces. PASS traces are composed of a series of records. Each record records a single operation, such as declaring a node, declaring an edge between two nodes, or annotating a node in some way. Annotations include adding a type (file, process, pipe, *etc.*), a name, a new version, a modification time for a version, and so on. Comparing our traces to the PASS traces allows us to confirm that the structure of our provenance graphs is similar to that of graphs collected by other means.

Table 5.3: PASS data sets

Trace	Description
`blast-chall-mirror`	A combined workload of Blast, 1st Provenance Challenge, and directory mirroring with tar
`chall-lite`	1st Provenance Challenge
`cp-lite`	A simple trace that involves Unix cp and cat commands
`elaine-oct25`	Researcher's desktop activity
`links-oct30`	A short session of provenance-aware Links (a text-based web browser)
`patch-apr17-2`	Patching the Linux kernel

5.2.1 Tests Performed

In order to evaluate our conversion algorithm, we performed a series of tests to examine how similar the structure of our created provenance graphs were to those collected by other means, and how the structure evolved over time. We replayed each trace forward in time to generate a graph, and compared the graphs at a sample interval based on the number of new nodes observed since the last sample. For longer traces, we compared them every 500 new nodes. For shorter traces which had fewer than $1,000$ nodes, we examined them at more frequent intervals to extract as much information as possible about behavior over time.

We compared graphs with equal numbers of nodes, up to $10,000$ files and processes.We examined the ratio of nodes to graph height. We looked at the distribution of branching factors. We also quantified the percentage of files and processes that we could not infer enough about to add them to the graph.

5.2.2 Details of Analysis

PASS creates a new version of a file or process every time new data is written to it. For purposes of analysis, we flattened these versions to get a holistic analysis of all accesses to a given process or file. PASS also does duplicate elimination, flattening all calls which would result in an identical edge being written to the database, *i.e.,* if a program writes twice to a file without any intervening data input, PASS treats those edges as the same. We do not do duplicate elimination, resulting in some edge count inflation relative to PASS.

PASS also tracks pipes. We noted these during analysis, and added them to the graph as a file node, but the LASR traces did not trace system calls for pipes, so we were unable to compare pipe statistics directly.

5.2.3 Errors

Like the LASR traces, the PASS traces did not always contain full information. For instance, there were a number of nodes which did not have a type attribute. In these cases, we used other information about the node. A node which is the result of a `fork`, or has an `exec` time or a process ID must be a process, and a node which has been unlinked or has an inode is a file. Despite inference, there are still some nodes about which we have no information. We calculated the percentage of nodes without type attributes to estimate the error rate of PASS.

Unlike missing information in traces, which can result in errors in the link structure of the output graph, omissions in the provenance traces cause errors in the annotation for nodes of the output graph. However, they do indicate that even a specialized tracing tool for provenance can have errors in the collection or output of traces, something which algorithms using provenance must take into account.

5.3 Results and Analysis

We estimate our coverage factor, based on observable error rates for process creation and file opens, and discuss how our graph compares to other graphs with regards to depth and branching factors. We find that our conversion tool has statistical properties similar to that of other system provenance graphs, and that our coverage, while slightly lower than other system provenance graphs, does not significantly alter the graph characteristics.

5.3.1 Coverage

Some errors are inevitable in any kind of data collection. If the omission rate is too high, it will affect the characteristics of the graph, making it less suitable for study. Thus, we examine the error rate.

The LASR traces have an average omission rate of 2% for file opens, and 8% for process forks. Recall that missing a file open results in us missing knowledge about read and write statistics for specific files, and may miss data flow relationships, whereas missing a process creation results in a "discontinuity" in the graph, where a new node is created without a link to its parent process.

The omission rate for missing process forks is consistently higher than that for file opens, which is common for many tracing architectures, especially those dependent on `PTRACE`. We assume that the omission rate for forks provides an upper bound for the omission rate of the whole trace (although we suspect that the true average is closer to that of file opens.) Thus we infer a 8% maximum omission rate.

While it is less common, it is worth observing that the traces from PASS also have missing information about nodes. On average, 0.5% of PASS nodes do not have a TYPE attribute, and we must infer a type based on their operations. Even after inference, we cannot determine a type for most of these nodes. Overall, 0.3% percent of nodes contain insufficient information to determine whether they are a file, a pipe, or a process, even after examining their operations. There is not enough information in the traces to know whether this 0.5% error rate holds for other operations as well.

We conclude that the error rate is moderate, and that the graph is 92% correct at minimum. Next, we examine the characteristics of the graph to see how much it is affected by the error rate.

5.3.2 Depths

Because of the nature of provenance graphs, height is not as simple to measure as in a tree. A node can start out as a root (a file which has never been written to or a process which has not read) and then later be written or read, becoming an internal node. Processes cannot accumulate reads or writes after they exit, but some processes are long running, so may continue to acquire edges for some time, and any undeleted file can continue to accumulate edges. And there are many possible paths to a given file or process. Thus it is best to think of provenance graph height as the maximum or minimum length of the path that data can have followed, which may grow or shrink over time.

We measure the height from the *leaves*, files and processes which have not been read, or have not written, respectively. Leaves can change over time, and should be considered as the periphery of the graph. From the leaves, we measure both the shortest and the longest path to a given node (breaking cycles as we go, since provenance graphs are not guaranteed to be acyclic), giving it a maximum and minimum height respectively. We examine the average and maximum values for both of these heights. (The minimums are, of course, zero, since leaves are guaranteed to have both a zero minimum and maximum height.)

As we can see from Figures 5.2, 5.3, 5.4, and 5.5, the graphs are quite similar in heights for a given number of nodes, particularly the average minimum and maximum heights in Figures 5.3 and 5.5, respectively. This suggests that the small percentage of orphaned processes and nodes in the LASR traces do not significantly alter the overall graph structure.

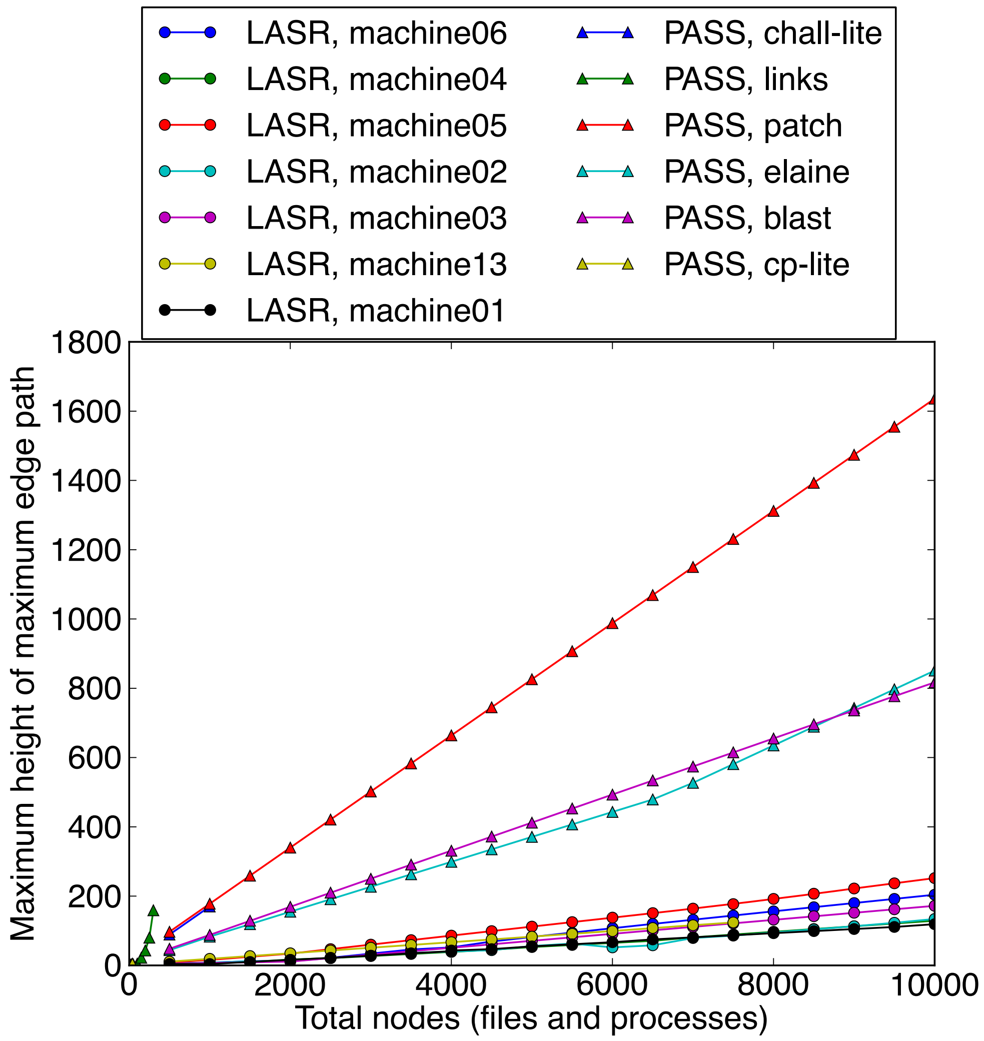


Figure 5.2: Tallest maximum heights. PASS traces have taller maximum heights.

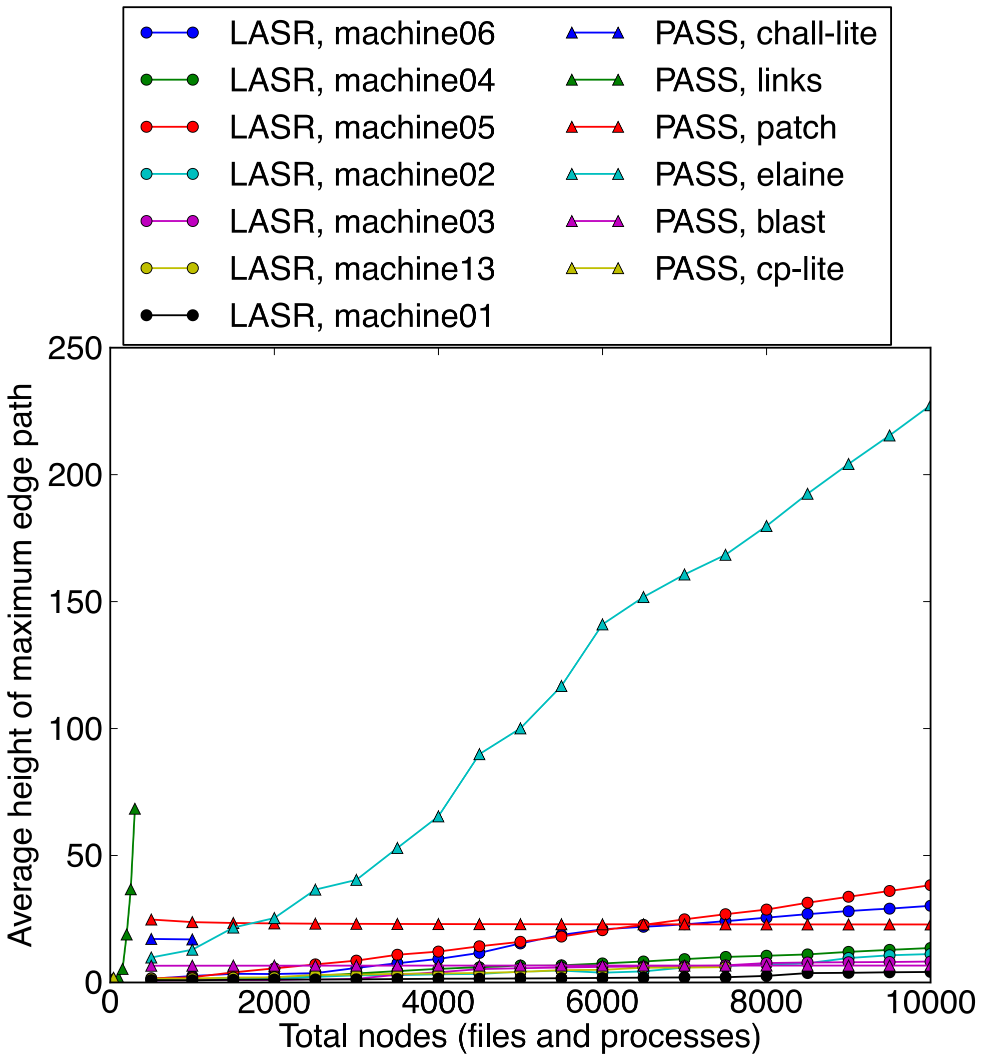


Figure 5.3: Average maximum heights are quite similar for LASR and PASS traces.

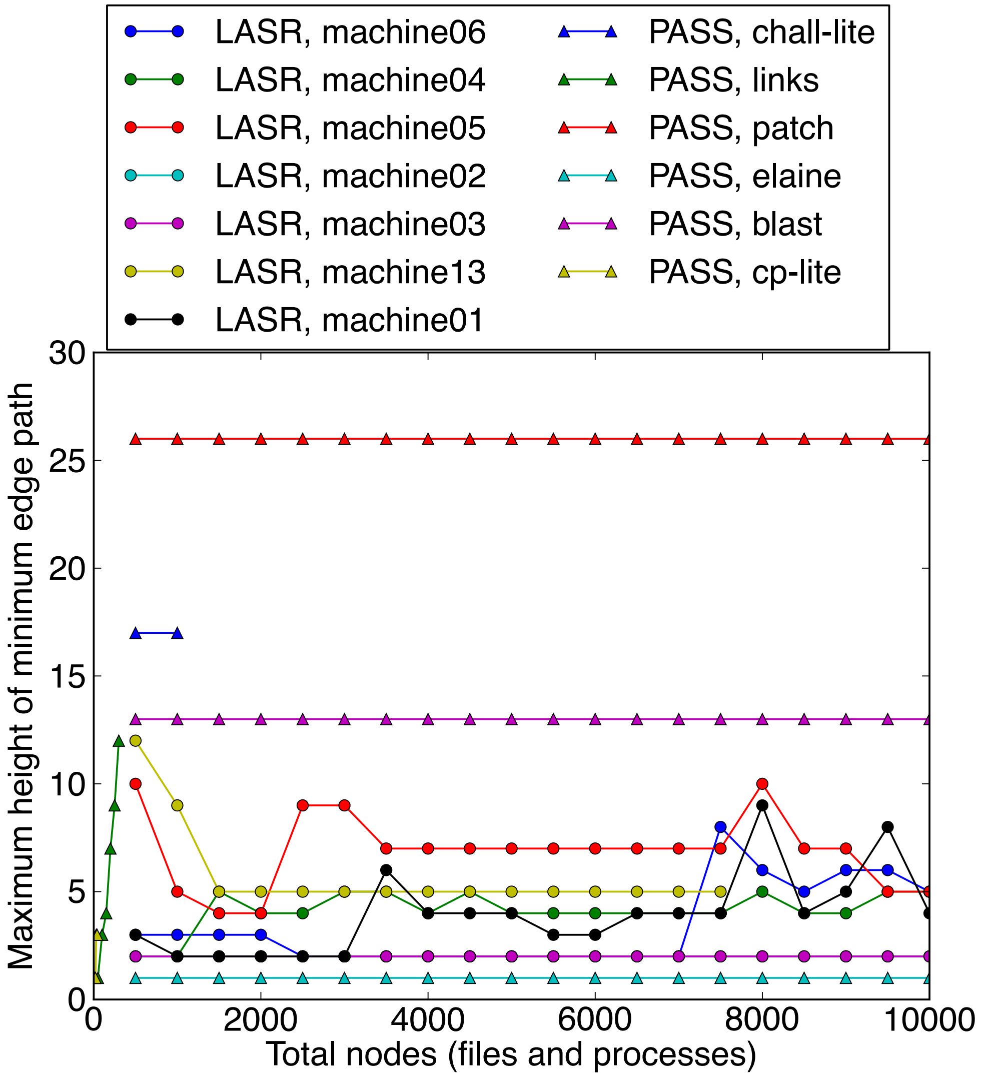


Figure 5.4: Tallest minimum heights vary significantly, but the LASR traces fall between the extremes of the PASS traces.

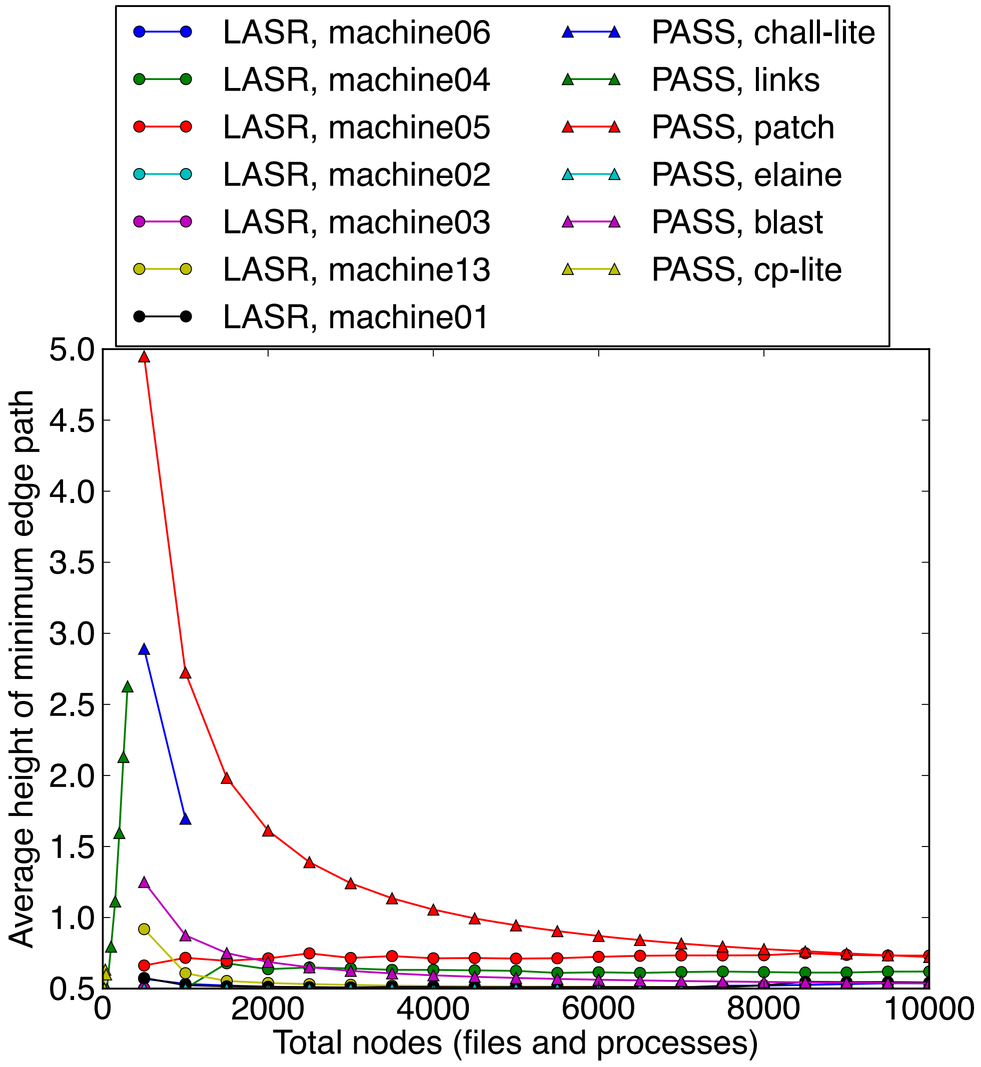


Figure 5.5: Average minimum heights are quite similar, and converge among most traces.

5.3.3 Branching factors

Next we measure how many times data flows in or out of a file or process, as shown in Figures 5.6 and 5.7 for inflows, and 5.8 and 5.9 for outflows. This can vary significantly by workload. For instance, a single `untar` can create a very large number of outflows from a process. The LASR traces are primarily personal computer workloads, with a mix of programming, web surfing, and so on. We compared them to all the PASS traces, but the `cp-lite`, `elaine`, and `links` traces are the most similar workloads. Unfortunately, `cp-lite` and `links` are very short, making it difficult to draw useful conclusions. However, we observe that most of the traces for both workloads have relatively similar branching factors, but that there is a great deal of variation across different traces and at different times.

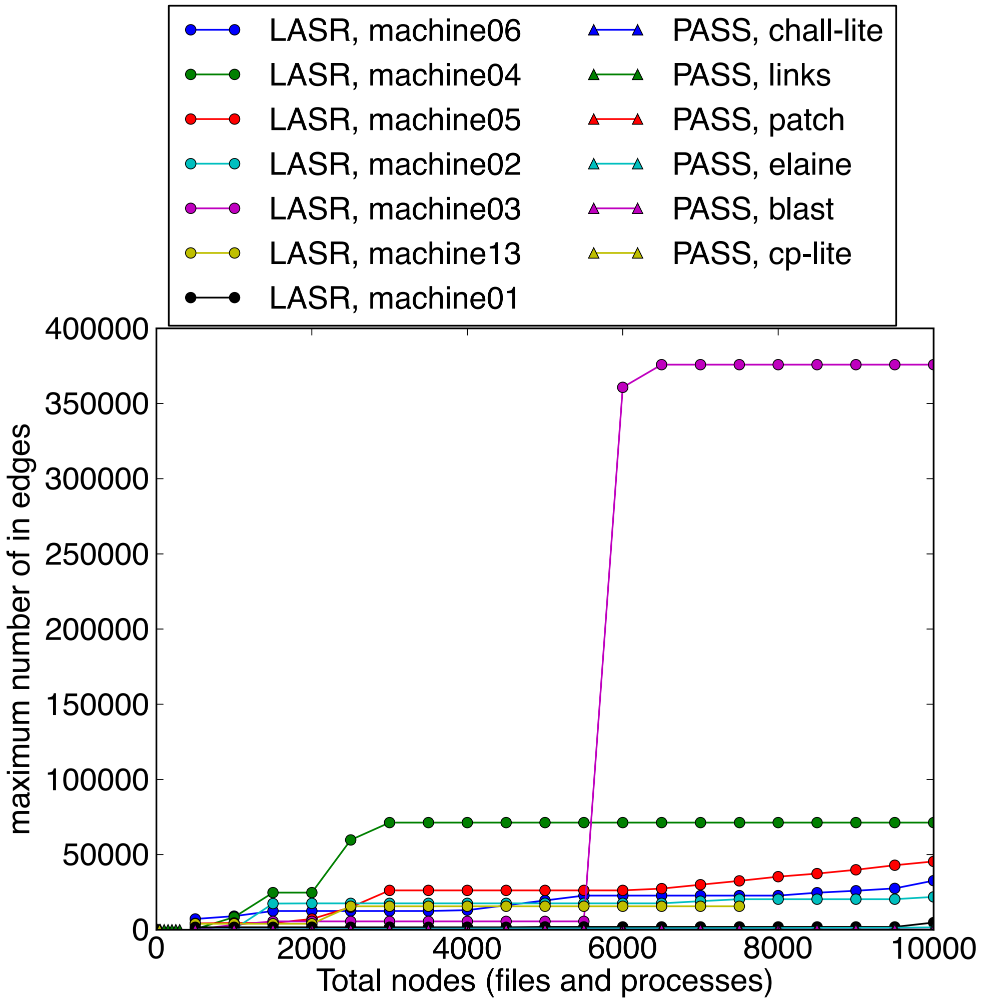


Figure 5.6: Maximum data flows into a node. LASR traces show evidence of non-deduplication, but are similar to PASS traces.

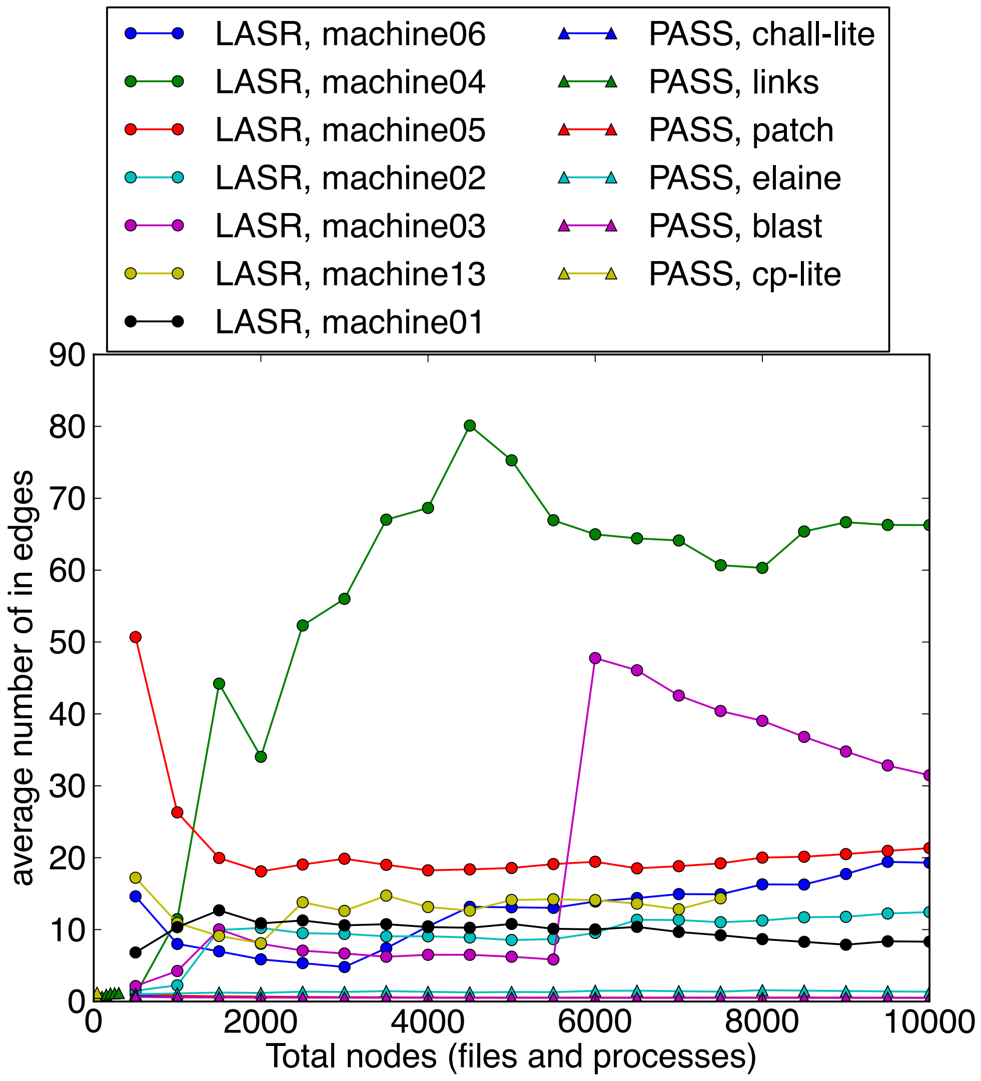


Figure 5.7: Average data flows into a node. The average is very noisy, suggesting that a few outlier nodes account for the bulk of most data flow.

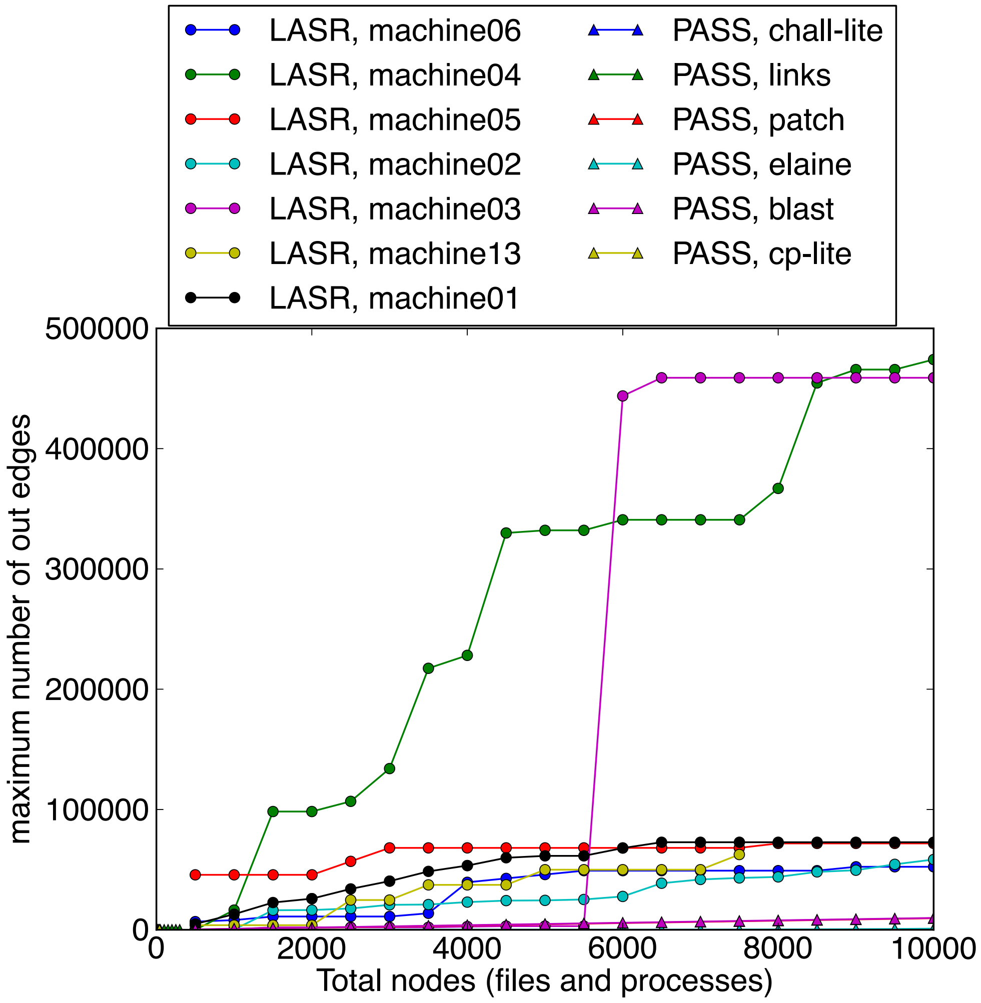


Figure 5.8: Maximum data flows out of a node. LASR traces show evidence of non-deduplication, but are similar to PASS traces.

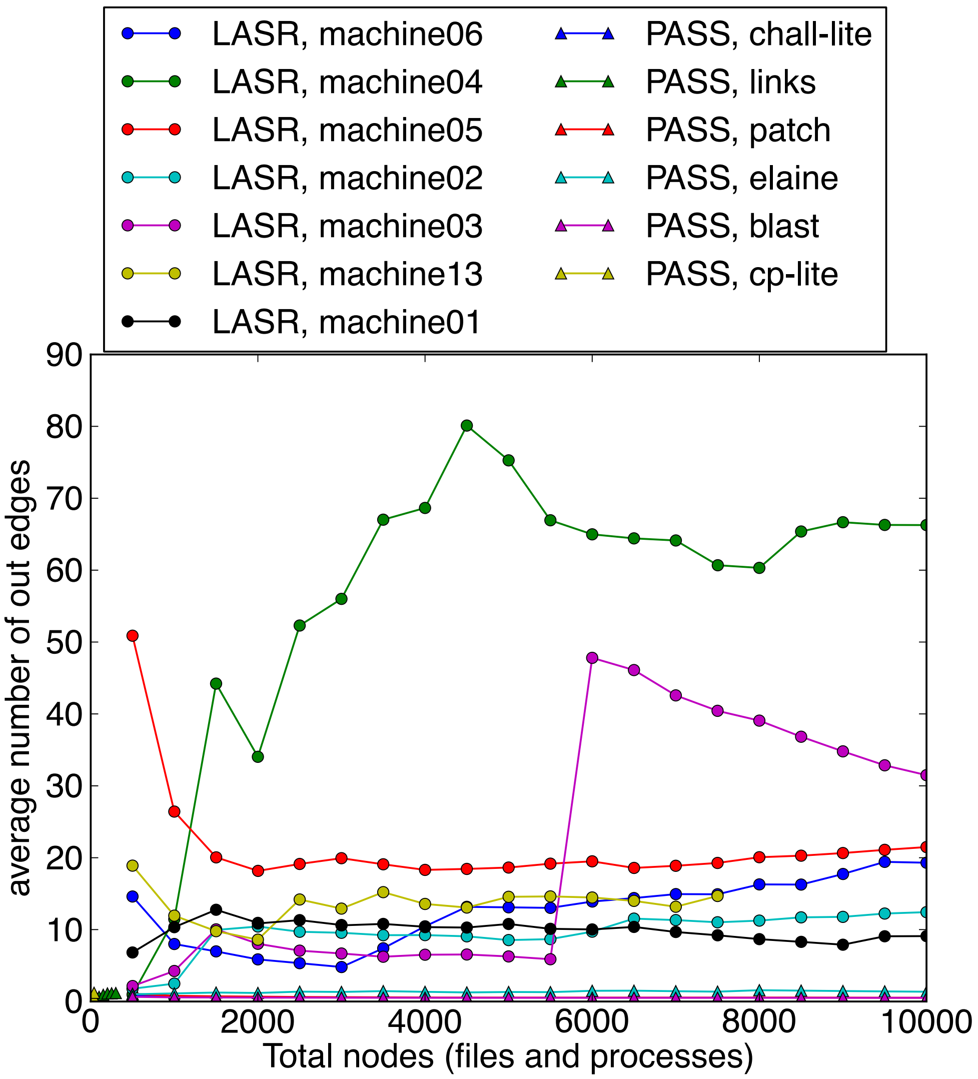


Figure 5.9: Average data flows out of a node. The average is very noisy, suggesting that a few outlier nodes account for the bulk of most data flow.

5.4 Conclusions

Our technique for creating provenance graphs from existing file system traces is novel, and provides access to new sources of provenance information. Our technique is feasible, as we have shown on the LASR traces, and is comparable to other provenance collation mechanisms, as we have shown by comparing it to existing provenance traces created by PASS. Our technique achieves high coverage, at not less than 92% of system calls, and has graph properties similar to other forms of file system provenance collection. Creating provenance graphs in this way allows us to leverage the wealth of existing file system traces, in order to study provenance over longer time periods than is currently possible. The provenance traces we have created compose the single largest set of provenance graphs to date, spanning almost a year, instead of a few hours or days. This makes them an invaluable resource for provenance research.

Chapter 6

Predicting Access Through Provenance

I count the grains of sand on the beach
and measure the sea;
I understand the speech of the dumb
and hear the voiceless.

Herodotus–The Pythian prophetess at Delphi

Having described how to create provenance graphs from existing data, we turn to the question of using provenance to improve search.

We describe a method for performing ranked access prediction, *Time and Provenance Informed Ranking (TaPIR)*, which creates a predictive distribution based on an annotated provenance graph, such as those created by our tool. This may seem counter-intuitive, since there are a number of prediction algorithms available which use smaller amounts of data. However, provenance data has a number of advantages. The first is that it's already in high demand by file system users, especially in the high performance computing world, who have the largest amounts of data and the most pressing need for better search. If the data is already going to be stored, it makes sense to leverage it, rather than storing additional data to serve the same purpose. Additionally, provenance offers a richer structure and annotations, which can potentially be leveraged into further access prediction applications, such as personalized search predictions, caching dependency trees based on sub-graphs, and so on. We compare provenance based access prediction to a number of other algorithms, and demonstrate that it can be used to create better predictions than other algorithms, while not requiring additional storage beyond provenance data.

Ranked access prediction has a number of applications. High quality access prediction at the file level is useful in its own right. For instance, being able to accurately predict file accesses can be used to improve caching in multi-tiered file systems, as well in other domains where parts of the storage system may be slow, or unreachable. (For instance, on a cloud attached laptop, or a mobile phone which has intermittent network connectivity.) Caching whole files can significantly improve the user experience in these cases, where caching at the block level will result in poor usability and unintuitive failure modes.

In addition, we argue that in the file system domain, there is a clear connection between access prediction and ranking for search results. If we have an oracle, which can perfectly predict the distribution of files the user will open next, then that oracle can also be used to rank search results, since the best search results will be the ones you will open next. Thus doing better quality access prediction for file systems is a useful step towards creating better file system search at scale.

Results ranking is a mature field, with two primary components. The first one, that of *similarity*, is extremely well understood for text content, and is actively researched for other content types such as images and music. However, in this work, we focus on the second component, that of *importance*. Importance ranking is designed to rank files, all of which match a query and might be suitable answers, based on their importance and likelihood of selection.

In order to do importance ranking, we need to know what is important. However, that's a difficult question to answer, even for a human. Ranking studies with human relevance judgements are hard and expensive, and even experts in an area often will not agree on the importance of a search result [99]. So surrogate metrics on large amounts of activity data are commonly used. On the web, the measure of importance is whether a page is *linked to* by important pages, whether you *click* on a search result, and whether you *linger* on a search result, or immediately return to Google and try a different link. On a file system, by analogy, we might want to ask the question what are the *relationships* between files, what do you *open*, and what do you *read*? This is exactly the data stored by provenance.

On the web, importance ranking is performed by a large number of algorithms, commonly used in ensemble, the most famous of which is PageRank [79]. PageRank uses a Markov chain based on the link structure of the web to generate a probability distribution that a given page will be important.

Likewise, TaPIR is a Markov chain based algorithm. However, unlike PageRank, TaPIR uses the provenance graph structure, particularly reads, along with a time decay function to generate a probability distribution over all files, predicting the probability that a given file will be opened. We show that TaPIR can outperform a variety of other access prediction algorithms, such as last-k accesses, all-previous-accesses, and time decayed access counters.

6.1 Architecture

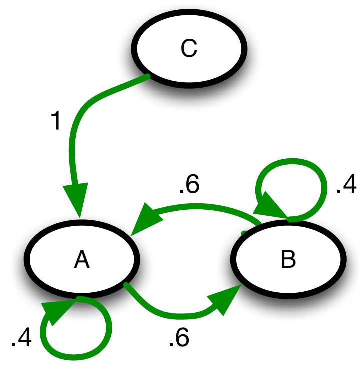


Figure 6.1: A Markov chain with three states

TaPIR is designed to operate on the structure of of a provenance graph by converting it into a Markov chain and generating a predictive distribution for file accesses. What we use as an access predictor is the *stationary distribution* for the Markov chain, the probability of being in a given state at a given moment.

Recall that Markov chains are composed of a set of states, and probabilistic transitions between those states, as seen in Figure 6.1. The probabilities of transitions are fixed, and depend only on the current state. Markov chains can be represented as a stochastic matrix where the rows and columns each represent nodes, each row sums to 1, and a given element $E_{i,j}$ is the probability of transitioning between node i and node j.

We treat the files in the provenance graph as the states of the Markov chain, and edges as transitions. If file F_a is written by a process P_i after that process has read file F_b, we consider that evidence of possible data flow, and create a *ancestry* transition, $E_{b,a}$ from F_b to F_a with a transition probability $\Pr(E_{b,a})$.

Markov chains have stationary distributions if and only if the following conditions are met:

1. The Markov chain must be irreducible, *i.e.,* it must be possible to get from any state to any other state given some finite number of transitions.

2. The Markov chain must be positive recurrent, *i.e.,* if we start in any state $S_i \in S$, we expect to return to it in some finite amount of time.

Once the first holds true, the second naturally follows for any Markov chain with a finite number of states, such as our provenance graph. One method of ensuring the first condition holds true for a general graph (such as a web or provenance graph, which are generally not irreducible), is to introduce a *teleport* function. This adds a non-zero probability of transition, denoted as α, from each state into every other state, as shown in Figure 6.2. This ensures that we can always get to any state to any other state in a finite amount of time with non-zero probability, satisfying the positive recurrent condition.

We use a weighted non-uniform teleport function, and apply a exponential decay function to discount the weight of older files, with a decay parameter, λ. This leverages previous research about the relative predictive value of accesses over time. The older the previous access, the less relevant to future accesses [104]. We also apply the same discount when weighting ancestry relationships between files.

We calculate the weight of a given node, F_i as follows:

$$\text{mtime}(F_i) = \max(\text{edge time})\ \forall \text{edges} \in F_i$$

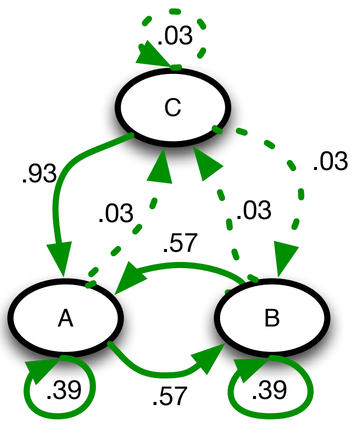


Figure 6.2: A Markov chain with three states and a teleport probability of $\alpha = .1$

$$\text{age} = \max(\text{current time} - \text{mtime}(F_i), 1.0)$$

$$\text{weight}(F_i) = e^{-\lambda \times \text{age}} \times |\text{reads}(F_i)|$$

Once calculated, we normalize all the weights to sum to α, our teleport probability.

We then calculate the transition probabilities of the Markov chain based on our weights. The unnormalized probability of transitioning from a node F_i to a node F_j is denoted as

$$\Pr(E_{j,i}) = |\text{reads}(F_{i,j})| \times \text{weight}(F_j)$$

So if F_i has read from F_j 4 times, and F_j has a weight of 0.001, then $Pr(E_{j,i}) = 0.004$ before normalization.

Once we have calculated all of the unnormalized probabilities, we normalize them before adding the teleport probabilities, such that

$$\forall i, j \in F, \sum_{e \in F_i} \Pr(e_{i,j}) = 1 - \alpha$$

Finally, we add the transition probabilities, weighting the teleport according to the weight of the destination.

$$\forall i, j, E_{j,i} += \text{weight}(j)$$

If F_i has outbound edges, then

$$\sum_{e \in F_i} \Pr(e) = (1 - \alpha) + \alpha = 1$$

If F_i has no outbound edges, *i.e.,* $\sum_{e \in F_i} \Pr(e) = 0$, then this is not a proper probability distribution, since it sums to α, rather than 1, so we normalize one last time to ensure a proper stochastic matrix.

$$\forall i, j \in F, \text{ if } \sum_{e \in F_i} \Pr(e) = 0, \text{ then } E_{j,i} = \text{weight}(j)/\alpha, \text{ such that } \sum_{e \in F_i} \Pr(e) = 1$$

Having created a stochastic matrix using the provenance graph and node weights, we then need to find the stationary distribution, the overall probability distribution for all files.

There are several methods for finding the stationary distribution of a Markov chain, which generally vary only in efficiency. However, the most popular one is to apply eigenvector decomposition, which has a number of very efficient implementations. The stationary distribution is by definition a row vector π, which satisfies the equation $\pi = \pi E$. In other words, multiplying the stationary distribution π by E, the transition matrix, does not rotate it nor scale it. This is the definition of a left eigenvector whose eigenvalue is 1. Thus, we can apply eigenvector decomposition to the matrix, and take the left eigenvector whose value is 1 to find the stationary distribution. We then take that as our final predictor for file accesses, after normalizing it to sum to 1. (We note in passing that this is the same method used by PageRank [79].)

6.2 Experimental Design

To evaluate TaPIR, we compare it against a variety of other predictive distributions, which we also describe, along with our experimental methodology.

6.2.1 Comparison algorithms

To evaluate our predictor, we compared it both to an oracle, and to a variety of other algorithms. We compared it to a time decayed access counter, which discounted the reads of the file based on the most recent access time of the file. This uses the same formula as the weights above, but we also evaluated it in isolation, since it is similar to previous work in access prediction, and we tested a variety of values for λ, to achieve best performance. We also compared it to a probability distribution based on the previous N system calls, as well as the un-truncated probability distribution of all previous accesses.

6.2.2 Windowed prediction with an oracle

As our oracle, we used the next 1000 system calls to generate a distribution of the actual file accesses. We then compared our generated distribution from TaPIR with the true distribution of accesses every 1000 lines, using Spearman's rank correlation coefficient [91].

Spearman's rank correlation coefficient is used to calculate the correlation between two ranked lists of items. It is commonly used in evaluating ranking functions for search, since the actual value of the rank is less important than the ordering of results, and the function penalizes rankings according to how far down the list they have moved from their true ordering. A perfect ranking function will have a correlation of 1, and the worst possible ranking function will have a value of -1. Spearman's rank correlation coefficient is defined as:

$$\rho = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2}}$$

where x_i is the rank of the ith item in x, and $\bar{x}$ is the average rank of all items.

Since we are dealing with file rankings, we order a list of file names by their access frequency (either predicted or actual). Spearman's rank correlation coefficient requires both distributions to have a matching number of categories, and the actual accesses may contain new files which we have no prediction for. Thus we apply a smoothing function, which assigns a very small value to any unseen file (the minimum of of all predicted values divided by .001, or the minimum float for the system, whichever is larger.)

Since we are using smoothing, which ranks any unseen file at the bottom of the prediction list, a value of -1 often means that new files account for the majority of accesses. A value near 0 suggests that the predictions are randomly correlated with the actual value, and the closer to 1, the better the predictions.

6.2.3 Parameters

The canonical value for α in web ranking is 0.15. This parameter can be used to trade off the quality of the final ranking with the speed of convergence [47]. However, the degree to which it affects ranking depends on the structure of the underlying graph [47]. We varied the α parameter from 0.01 to 1, and noted that for our graphs 0.15 did have the maximum performance for ranking. Thus, we set $\alpha = 0.15$.

We also evaluated values of λ from 0.000001 to 1000, for both our time decayed access counters, and our node transition weights. The access counters performed best at 0.000001 and 0.00001, depending on the trace, and TaPIR peaked at 0.00001, so we use both of those values in all the evaluations. These numbers suggest that for these traces, the working set decays slowly.

6.3 Results and Analysis

Our experimental results show that TaPIR performs as well as our comparison predictors, and often better. It significantly outstrips the predictor using the previous N accesses, and generally beats time decayed access counters. The closest in performance is the predictor which uses all previous accesses to build a distribution, which TaPIR slightly outperforms.

We evaluated our algorithm as well as our comparison algorithms on several different traces from the LASR dataset, as seen in Figures 6.3, 6.4 and 6.5. We recalculated the predictions for all files every 5000 system calls for each algorithm, resulting in 39 data points for each graph. We compared the performance of our algorithm to the other algorithms by examining both the average improvement in correlation, as well as the statistical significance of our improvement, as captured in Table 6.1. The Mann-Whitney U statistic is a non-parametric test which compares the total rankings of two distributions to examine the likelihood of them being drawn from the same distribution. When U is near $n \times n/2$, i.e., when each algorithm has beaten about half of the other algorithm's data points, we accept the null hypothesis that they are the same distribution. For 39 data points, our critical value is at $39 \times 39/2 = 760$.

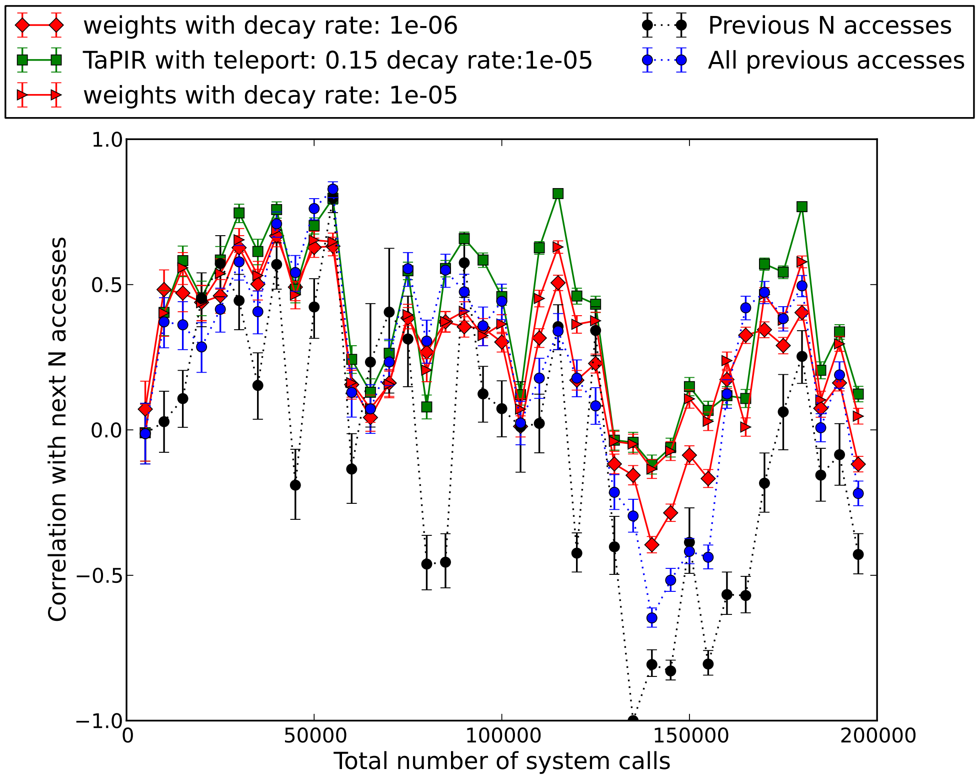


Figure 6.3: Machine 1: Provenance based prediction results with comparisons. TaPIR performs comparably with all of the other algorithms.

All algorithms perform poorly with very little data, so early predictions are naturally unreliable, as seen in Figures 6.3, 6.4 and 6.5. As more data enters the system, all of the ranking algorithms begin to improve. Once warmed up with sufficient data, TaPIR performs as well or better than the one based on all previous file accesses ($0.0275 \leq P \leq 0.2421$) and access counters ($0.0133 \leq P \leq 0.2484$), and significantly better than a predictor based on the previous N files ($0.0 \leq P \leq 0.0001$). All of the algorithms are correlated in performance, indicating that the workloads varies from highly predictable to unpredictable, but that TaPIR makes the best use of available data. We conclude that TaPIR can be used for ranking and access prediction, allowing us to leverage provenance data for both of these tasks.

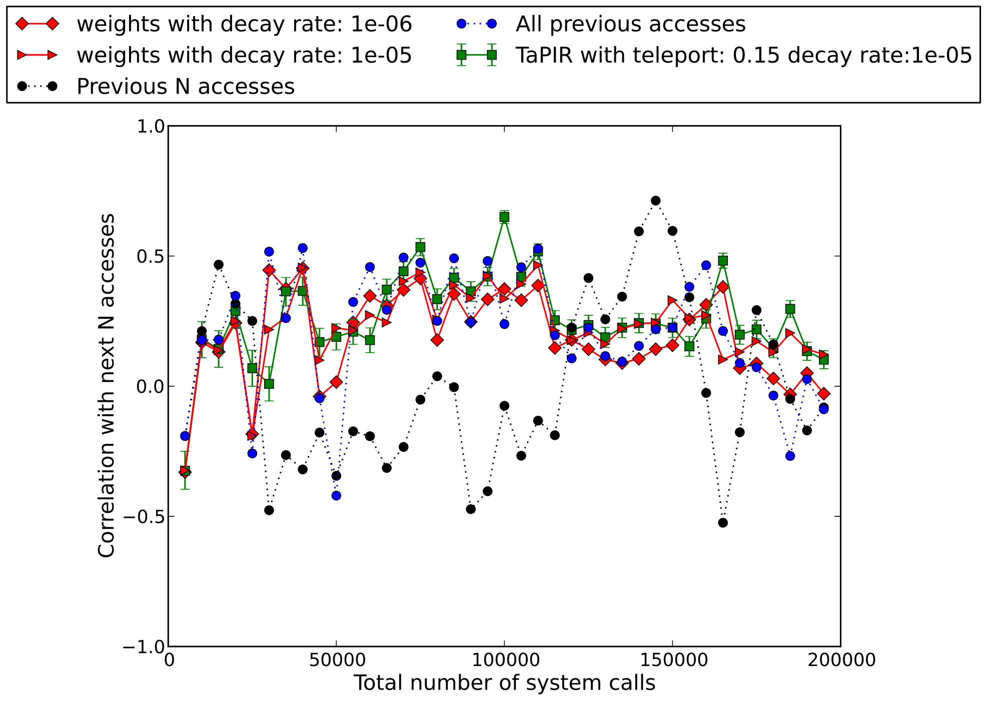


Figure 6.4: Machine 2: Provenance based prediction results with comparisons. TaPIR performs comparably with all of the other algorithms, although less well than on Machine 1. Accesses starting at 157722 are dumping the trace to a single file.

Table 6.1: TaPIR Improvement Rate and Statistical Significance

Machine	Algorithm	Improvement	Mann-Whitney U	p-value
Machine 01	Previous N accesses	0.4205	333.0	0.0000
	Weights ($\lambda = 10^-5$)	0.0755	569.0	0.0281
	Weights ($\lambda = 10^-6$)	0.1353	616.0	0.0751
	All previous accesses	0.1617	568.0	0.0275
Machine 02	Previous N accesses	0.2619	394.0	0.0001
	Weights ($\lambda = 10^-5$)	0.0335	586.0	0.0410
	Weights ($\lambda = 10^-6$)	0.0711	692.0	0.2484
	All previous accesses	0.0540	690.0	0.2421
Machine 03	Previous N accesses	0.3080	373.0	0.0001
	Weights ($\lambda = 10^-5$)	0.0684	523.0	0.0133
	Weights ($\lambda = 10^-6$)	0.1361	603.0	0.0806
	All previous accesses	0.2270	379.0	0.0001

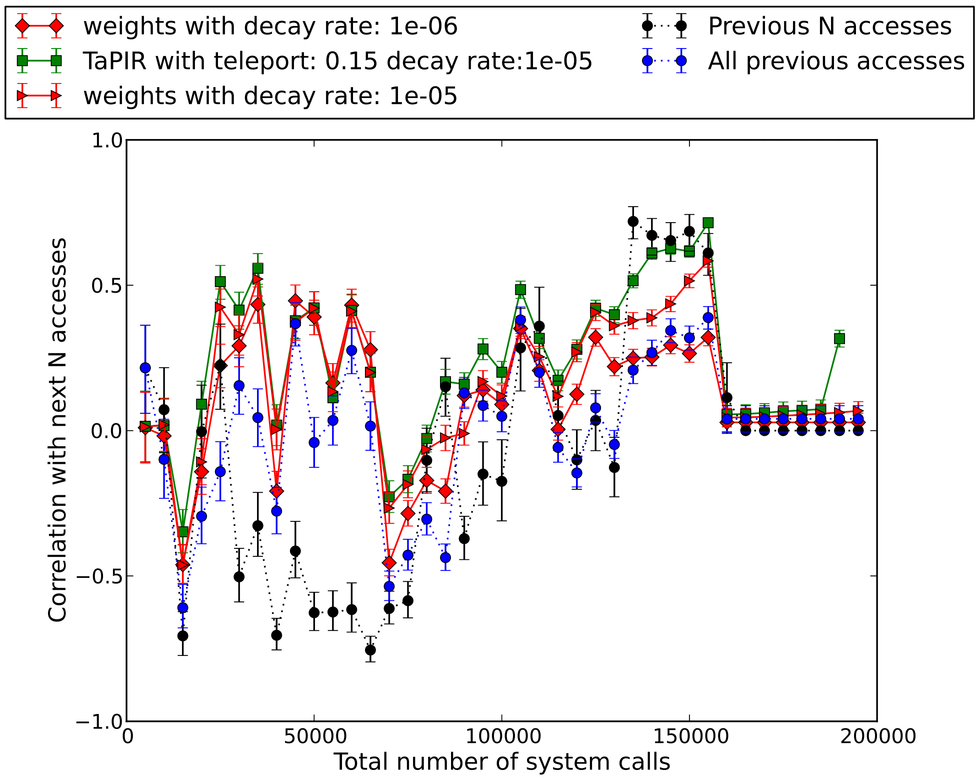


Figure 6.5: Machine 3: Provenance based prediction results with comparisons. TaPIR performs comparably with all of the other algorithms.

6.4 Conclusions

A key tool for scientists, file provenance, can also be leveraged to perform file ranking and access prediction, both of which improve file management. Provenance based ranking performs comparably with other access prediction methods, such as last-k accesses, and time decayed access counters. Provenance based ranking also does not require additional storage beyond provenance data. Additionally, provenance contains a wealth of information, such as workflow graphs and user information, which can potentially be leveraged for more personalized ranking, opening up an exciting new area of future research on improving predictions further.

Scientists, more than anyone, need better file system search which allows them to quickly find important files. Utilizing already existing scientific data to improve caching and search will significantly improve the state of scientific data management. Our method uses data that scientists have already expressed need for, and applies it to the problem of creating better caching and search for scientific data.

Chapter 7

Metadata Aware Naming

Stat rosa pristina, nomine, nomina nuda tenemus
"Yesterday's rose endures in its name, bare names are all we have"

Umberto Eco, *Name of the Rose*

File names have existed since the earliest file systems, and serve two important functions. First, they serve to uniquely identify a file over time. Second, they serve to jog our memory, describing the contents of a file, and helping us to find it or recognize it when we look at it later. In order to help users to find and remember files, they often contain a bounty of useful metadata about the file.

However current approaches to file names have flaws:

- They are unstructured semantic metadata which is opaque to the system
- Formatting is up to the user, and is error prone and inconsistent, making it hard to find files later
- Changing a file name destroys information

Consider the following file name drawn from the author's experiments:
`createfiles_HDD_truenames_100000files_1threads.data`
Looked at in one light, this is a long, arbitrary, error prone string of characters. In another light, it is a rich source of semantic metadata about the contents of the file that could be used for search and analysis. The challenge is to extract that information.

To resolve these flaws, we propose to disassociate the two functions of names, separating the task of uniquely identifying a file for applications from helping users to identify a file. As a proof of concept, we describe our new prototype file system, TrueNames, a FUSE-based file system which provides a durable unique identifier for a file which can be used by applications, captures rich metadata in a structured format, and uses it to dynamically generate user-friendly file names using *templates*. These file names have many advantages over conventional file names. They are *correct*, because they are continuously synchronized with the file's metadata. They are *regenerable*, allowing us to compress and recreate names at will without loss of information. File names can be *disambiguated* using all available metadata, which reduces accidental data over-writes. The metadata which we capture is structured, making it readily available for search and data management. We describe these new file names as *metadata aware* names.

TrueNames is designed to demonstrate the feasibility of several goals. Metadata aware file naming can make naming more reliable and less error prone. It can also prevent accidental overwriting of existing files by detecting collisions between names, and adding additional metadata to disambiguate. Many applications already offer some form of automatic file naming, and by making that functionality part of the file system, we can speed application development, and prevent code duplication and the resulting proliferation of bugs. Being able to regenerate file names allows us to port files between file systems with differing constraints, generating a meaningful file name in each location, and then reconstructing the original file name whenever needed. We can easily move metadata between file names and directory names, or store it for later use. And finally, by gently encouraging users and applications to share more metadata in a structured fashion, we can make metadata more readily available to improve the quality of file system search or support a non-hierarchical file system.

While TrueNames is a user space prototype file system, it demonstrates the feasibility of doing metadata aware naming. It offers extensive new functionality, and incurs very low overheads, less than 15% on realistic workloads. Much of the additional cost is incurred by added kernel crossings, suggesting that an in-kernel implementation would have negligible performance impact, while significantly improving file system search and data management by increasing the structured metadata available.

7.1 Use Cases

Metadata aware naming can be used as a broadly applicable framework for solving cross-cutting concerns. It can be used by applications to simplify common tasks, and by scripts and end-users to better manage files. It can even help to prevent data loss caused by overwrites, and allow multiple applications to cooperatively name files. We describe a variety of use cases for TrueNames, and explain how it can benefit users and applications in each case.

7.1.1 Managing experimental data

One of the common problems a scientist faces is that of managing experimental results. A scientist might run a series of experiments, varying some parameters while holding others constant, and outputting the results to a file, then decide based on the results to vary other parameters. This is commonly managed by creating file names programmatically. However, this approach has some drawbacks. For instance, if the user wants to search the results by parameter, they will need to use string matching or a regular expression, which relies on consistent formatting, such as using the same field separator and the same field order. Likewise, a metadata field which contains the same character as the chosen field separator (such as an underscore in a library name) can throw a regular expression off. Finally, common queries such as range searches require complex regular expressions to perform.

Another common problem occurs when rerunning experiments while setting additional parameters. The user also needs to remember to change the file name output to reflect the new parameters, and older file names will not have the new parameters, even if they were applicable. If the user forgets to change the code which generates file names for experimental results, they may overwrite existing results files, wasting hours or days of compute time.

TrueNames simplifies creating file names and searching for metadata. Rather than programmatically generating file names, experiments can simply add metadata for all the experiment parameters. File names can be generated using a simple template, and then the template can be expanded by the user as new parameters become relevant, which results in both old and new files using the new template. File name collision detection can help prevent overwriting by disambiguating files on the fly. The results files are easily searchable by metadata field, regardless of the order fields occur in the file name, or what characters occur in the field. Using structured data allows both exact match and range queries, and file names are an accurate reflection of their contents. The authors used TrueNames to manage experimental results for this work, and found it to be extremely helpful and simple to use, allowing them to easily manage multiple experiments and parameters, generate accurate and meaningful file names, and then search their results later.

7.1.2 Managing research papers

Applications such as Mendeley [7] are designed to manage a collection of research papers, making them easy to search. Mendeley indexes all of the metadata fields of each publication, does full text indexing, and can manage directory structures and file names. Mendeley does not use a fixed format for file names. Rather, they allow the user to pick the metadata fields and the order in which they should be used to generate a file name, much like TrueNames does. Applications such as Mendeley can work symbiotically with TrueNames, serving as an interface for extracting metadata and helping the user generate templates, while allowing TrueNames to do the work of keeping file names up to date.

7.1.3 Managing a music collection

Most modern MP3 players, in addition to playing music, also perform metadata and file management. They can download new music, organize music into folders, and assign names to music based on its metadata (stored in ID3 tags, an embedded metadata standard for MP3s [4].) For instance, iTunes will place MP3s into folders based on artist and album, and then generate a file name based on a fixed template using the disc number, track number, and song title. However, this means that music can only be managed through the application. New music downloaded from outside the application is unknown, and changes to music's metadata are not reflected in the file name unless done through the application. Utilities such as MusicBrainz Picard [8] can repair missing metadata and names, but require the user to recreate their iTunes database to reimport the new file names.

TrueNames, like iTunes, can manage file names based on metadata, using whatever fields are desired. However, file names will always be kept in sync with the latest metadata, regardless of the source of that metadata. New music can automatically be given a name in the desired format, no matter what application downloaded it. If the user prefers a different name, they can choose a different format, and all files of that type will automatically be renamed. By using unique identifiers rather than file names, databases don't have to be recreated when file names change. If multiple files are imported with the same name, rather than overwriting the existing files, TrueNames can check the metadata to prevent a collision which would result in losing one or more files. If the files differ in one or more metadata fields, TrueNames can generate differing file names which allow the two files to be distinguished.

7.1.4 Managing a photo collection

File names have lagged behind UIs in the photography field, making it challenging to find and manage photo files. While most applications offer sophisticated GUI photo management, many use the default file name generated by the camera, which contains only a per-camera sequence number, such as `IMG_655.jpg`, or perhaps a manufacturer and sequence number, such as `DSC_1967.jpg`. File name collisions are common. The latest version of iPhoto [5] names photos using the camera's generated name. Derivative files may be given a suffix based on size such as `IMG_655_1024.jpg`, or, in the case of a facial recognition thumbnail, an index corresponding to order of face discovery, such as `IMG_655_face1.jpg`. Photo names cannot be managed by the user, and offer very little information about their contents. Higher end applications such as Aperture [1] or Adobe Lightroom [6] allow the user to bulk rename files during import and export, using metadata such as EXIF fields and creation dates. However, these applications cannot keep file names in sync if metadata changes, making it difficult to manage photos in more than one application. For instance, a user might want to use facial recognition from iPhoto, while touching up photos in Lightroom. If the user wants to find a retouched photo of a certain person, they cannot search for it using metadata, and if they wish to put the information in the file name, they must painstakingly name the files manually.

TrueNames can significantly improve photo naming, by taking metadata extracted by the application and automatically constructing file names for photos based on metadata such as where they were taken, who was in them, and what size they are. If the metadata changes (such as a recognized face, or the addition of geo-tagging data), TrueNames will automatically update the file name to reflect the correct information. In addition, TrueNames allows multiple applications which can operate on the same files, such as iPhoto and Lightroom, to share responsibility for generating a meaningful name. Both applications can export metadata which can be used for naming and search, and TrueNames can manage the formatting and creation of file names, merging metadata from both applications into a single meaningful description of the file, something which is currently very difficult.

7.1.5 Recreating file names

One problem found in computing is that of file name portability. For instance, some file systems support longer names and paths than others. When moving files from one system to another, such as from primary to archival storage, or between two servers, file names can be truncated, losing important information, and making it difficult to find a file, even if it is brought back from archival. TrueNames can help with this, by allowing a file name to be regenerated based on the metadata. For instance, files being moved to archival storage can be updated to use a different name template with fewer fields, which fits within name constraints without truncating the file name at an arbitrary point. If the file is retrieved from archival storage back to primary storage, the original file name can be recreated without loss of data. Since structured metadata was stored and used to generate the original name, it can also be retrieved to search the archive, or help create meaningful directory names, in essence pivoting metadata between file names and directory names as needed.

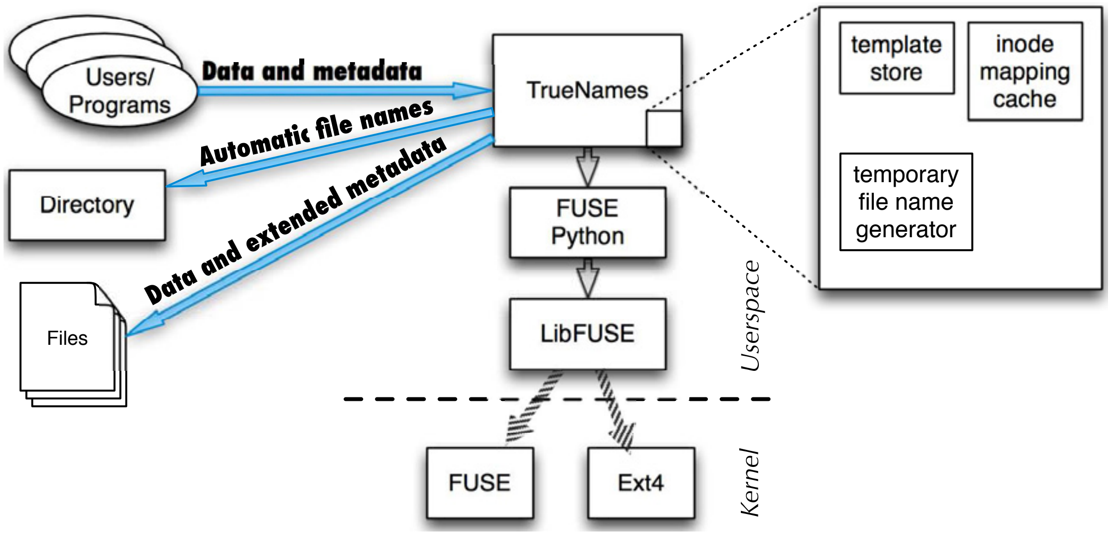


Figure 7.1: Architecture of TrueNames

7.2 Architecture

Having file names which can change outside the control of users and user applications poses a number of unique and interesting challenges. One must choose a storage mechanism for the metadata used for naming. There must be a way to define the structure of names, and what metadata they will use. Applications require a durable way to reference files, in order to maintain internal databases and repeatedly reference files. There must be a way to prevent accidental data loss through file name collisions. We discuss these challenges, and the necessary modifications to support metadata-aware naming throughout the file system and applications. Our goal for the prototype was to demonstrate a minimum set of semantic changes to POSIX to support metadata-aware naming. The architecture of TrueNames is shown in Figure 7.1.

7.2.1 Storing metadata

Our goal was to store rich metadata for file names in a way that was portable, did not significantly change the semantics of the file system, and was easy to understand and manage. To that end, we selected the extended attribute interface as being the most compatible with our goals. Extended attributes provide a simple key-value interface, and are associated with the inode (either by a reference to a metadata block or resource fork, or in the case of small metadata on ext4, directly stored in the inode), which means that files with hard links, which contain the same data, will also share metadata and names. (Soft links continue to offer the ability to have a mix of automatic and manual names for a single file.) Many applications already use extended attributes, they require no additional libraries, and are supported by most modern file systems, improving portability.

7.2.2 Managing names

Our file system assumes that file names have *schemas*, a set of rich metadata which is broadly applicable to many different files. However, not all files of the same type are assumed to have the same schema. For instance, an PDF file may be a scientific paper, a graph of experimental results, or an e-book. A text file might be a configuration file, or a letter to a relative. These will have different schemas that are appropriate. Likewise, two files of different types, such as `.jpg` and `.gif` files, may have a shared schema that is appropriate for both. Thus, we allow a *template* to be assigned to a file, which is a text string containing each of the metadata fields of the schema. A template defines the structure of a file name that is appropriate for that file, as shown in Example 1. Templates can contain file extensions which serve as the default extension. However, if the user supplies a file extension, it will override the template extension, allowing different types of files to share templates. For instance, `.jpg` and `.gif` files can share a `photo` template. When a `.gif` file is assigned the `photo` template shown in Algorithm 1, it will take the form `{$seq}_{$date}_{$camera}_{$location}.gif`, overriding the default `.jpg` extension.

Algorithm 1 Template file

```
music {$artist}-{$album}-{$track}.mp3
photo {$seq}_{$date}_{$camera}_{$location}.jpg
paper {$author}_{$conference}${year}_${tags}.pdf
exp {$wkload}_{$files}files_{$threads}threads.data
manual_name {$user.file_name}
```

In our prototype, templates are stored on a global basis, in a text configuration file, and loaded on file system startup, but more sophisticated implementations are possible. This file contains a list of template names and templates. Templates are associated with a file using an extended metadata field called `user.naming.type`, which references the name of a template. This field is not mandatory, making it possible to mix manually and automatically named files throughout the file system. If a user or application wishes to rename the file, they can change the value of the extended metadata field to the name of a different template, adding additional metadata as needed. If a user wishes to manually manage a name, they can remove the current template from the file's metadata, or not choose a template during file creation.

As metadata is added or updated by users and applications, the file name is updated to reflect the current state of the file's metadata. This entails storing the extended metadata, looking up the template, re-calculating the file name, and then renaming the file. It may also entail handling file name collisions caused by the rename.

If not all of the metadata is available at any point in time, TrueNames makes a best effort to update the file name, populating all of the fields that are known, and marking metadata that is missing with a default value (in our prototype, unknown metadata fields are marked with a `??`, but a per-template default is planned for future work). In our photo example, that might result in a file name of `DSC1967_7-21-13_NikonD50_{$location}??.jpg` for a photo that has not yet been geotagged.

7.2.3 File creation with automatic names

In order to create a file, we need a way to signal two things to the operating system. First, we must signal what directory we are creating a file in. Second, we need to signal what template we would like to use. Additionally, we must make sure that the file name we are creating is unique, to prevent overwriting an existing file. Finally, none of the extended attributes for the file are yet available, so we cannot use them to generate the initial file name. This is similar to the problems faced by `mkstmp()` and related functions. However, our goal was to maintain existing semantics as much as possible, and require minimal rewrite of existing code, which ruled out adding an additional system call. We therefore overrode the semantics of the existing system calls, `open()`, `creat()`, and `mknod()`, such that if a file is created with a name which ends in `template=X`, where `X` matches the name of a known template, the name is managed by the file system, and a new file is generated with an automatically generated unique file name. The initial file name is composed of the template contents, followed by a unique alphanumeric suffix such as `GzyH07`, and finally, any file extension, as shown in Example 2. We use atomic file creation with `O_EXCL` internally, to prevent race conditions, as well random back-off retries if we receive `EEXIST` during initial file creation. Once the file is created, metadata can be added to populate the file name. Since metadata must be added one field at a time, there is a possibility of a temporary name collision during metadata addition, which we attempt to detect and handle, as discussed in Section 7.2.5.

Alternatively, the application can create a file name using any name it wishes (including one given by the user), and then set the template and metadata fields after file creation. The original file name can be stored in metadata, and then reapplied later, or used for resolving collisions.

7.2.4 Programming with changing file names

One complication created by automatically named files is that the file name can change between accesses or modifications. For instance, when metadata is being added to the file, each new metadata item which is used by the template will trigger a rename. In these cases, the user or application needs a durable way to reference the file they are updating. TrueNames allows the user to reference files either by name, or using a unique identifier which serves as a durable reference.

In our prototype, we use the inode number to guarantee uniqueness of references. To guarantee that the inode has not been freed and reused, we will add the inode's generation number in the next version of TrueNames, similarly to how NFS handles stale file handles [89]. Unfortunately, most file systems do not provide a convenient way to look up and reference a file by its inode number. By contrast, we allow files to be referenced either by file name or directly by inode number, as if the inode number were a file name.

To reference a file in a directory by its inode number rather than by name, it can be opened using a reference to `<dirname>/.inode/<inodenumber>`, as shown in

Example 2. This allows programs and scripts to create an automatically named file, call `fstat` on the file handle to acquire a durable reference, and then add metadata, all without needing to know the file's user-friendly name.

Algorithm 2 Reference by inode

```
$ curl http://indyband.org/1.mp3 > template=music
$ ls -i
13146 {$artist}??-{$album}??-{$track}??nozwZ8.mp3
$ setfattr -n user.artist -v"Indy Band" .inode/13146
$ setfattr -n user.track -v"So Obscure" .inode/13146
$ setfattr -n user.tracknum -v"1" .inode/13146
$ ls
Indy Band-{$album}??-So Obscure.mp3
```

Under a typical file system such as ext4, it is not fast to access a file by its inode number. Accessing a file by its inode number requires searching the system for a file which has a matching inode number and then resolving it to a name, which can be prohibitively expensive. While we already reduce the cost by reducing the search space to a single directory, this still requires an linear scan over the directory inode. To get acceptable performance, we further reduced this cost by adding an inode cache map to TrueNames, which allows us to look up a file name in constant time given an inode number. This is similar to the name cache used by the Linux VFS, although the lookups occur from inode to name, rather than vice versa. The inode map is populated from the directory inode on first access, and then caches all inodes in that directory up to some threshold (ten thousand files in our experiments.)

By encouraging applications to use a static reference, while allowing the name to vary, users can modify file names at will to create more user-friendly names, without breaking existing application references. In addition, references by inode will work for all files, not just automatically named ones, so even manually named files can benefit from this feature.

7.2.5 Handling collisions

One problem that can arise with automatic naming is that two files in the same directory may be different in content and metadata, but share all the fields that are currently referenced in the template. During rename, we check to see if the new metadata results in two files with the same name. If so, we check to see if any metadata differs. If disambiguating metadata is available, we add the first available metadata to the file name which will disambiguate the files, along with its extended metadata key. If not, then and only then, do we overwrite the existing file. In the future, we plan to explore more user-friendly strategies for disambiguation.

7.2.6 Application level support

A file system which requires extensive changes to applications is unlikely to see adoption. By maintaining POSIX compliance, TrueNames makes adoption easier. Existing applications which don't wish to take advantage of the added functionality can run on TrueNames without any modifications, and see very little change in performance. In order to take advantage of the new functionality, applications simply need to create a template, and begin exporting metadata for each new file. Optionally, they can begin referencing files using an inode reference. We describe below the changes to semantics in our prototype, and how they affect applications.

`open()/creat()/mknod()` As noted above, if these calls are invoked using a directory path followed by a template name, they will create an automatically named file using the template as a name, followed by a unique suffix and and an extension, and set `user.naming.type` to the template name. If the path supplied does not end in a template, they will create a file in the normal fashion.

`setxattr()/removexattr()` In addition to setting and removing extended attributes, these calls now additionally trigger a recalculation of the file name. If the attribute set or removed is one present in the file template, then the file will be renamed. Additionally, these can be used to change the template, or even remove the file from the set of dynamically named files and freeze the current name, by setting or removing the `user.naming.type` extended attribute.

`rename()` This operation can be used in a variety of ways.

- If the supplied target path contains a different directory, but the same file name, the file is moved to the new directory using its current file name.

- If the target path contains a different file name which is the name of an existing template, we update the file to use the new template.
- If the target path contains a different file name which is not the name of an existing template, we assume the user wishes to manually control the file name. We rename the file to the new file name, and remove `user.naming.type` from the file's metadata.
- In all cases, if a new inode is created during rename (for instance, if the file is migrated between file systems to a file system which supports extended attributes), all the extended metadata is copied to the new inode.

`link()`**/Calls which use hard links** Under our prototype, a hard link shares an inode, and therefore all extended metadata, with the path it links to. Therefore, files which are hard links to an automatically named file will share a name with the file they link to. If a file is dynamically named, a hard link in in the same directory is not feasible, since it has the same name as the target. An attempt to create a hard link in the same directory will fail with `EEXIST`. Otherwise, hard links function as expected. This is a limitation of our prototype. In future work, we plan to create a semantically complete method for automatically managing both hard and soft links which will permit the file to have multiple names.

`symlink()`**/Calls which use soft links** Soft links work as before, and can target either a file name, or an inode path, depending on the desired behavior. Soft links can be used to supply multiple names in the same directory, such as an automatic name and a manual name. Due to restrictions on extended attributes, soft links cannot be automatically named in our prototype.

Every file now has at least two names: its human readable name, either autogenerated or assigned by a human, and its inode number preceded by its directory path. It may have additional names via hard and soft links. Human readable names and inode references are interchangeable in all file system calls. A reference by inode can be opened, linked, have metadata set or gotten, and so on. However, `.inode/` itself is not a real directory on disk, and cannot be opened or have its directory entries iterated over.

If multiple applications manage the same files, then there is the possibility of both applications attempting to manage the template and metadata. Adding additional metadata to file names and templates allows richer search, and can improve generated names, but applications may wish to prompt the user before changing the template or removing fields from it.

7.3 Experimental Design

We have demonstrated how new functionality can be added to the file system to make it more searchable, to make file names more correct and structured, and how this functionality can easily integrate into existing file systems. However, a file system's functionality must also be balanced against its performance. We describe how we evaluated the performance of TrueNames. In order to effectively benchmark such a file system, we need to answer questions on how it affects basic file system operations, such as file creation, deletion, and renaming. We also need to describe the effect on extended metadata operations. In order to evaluate our file system, we compared it against two other file systems, one comparable Python FUSE file system, as well as a raw ext4 file system in order to characterize the overhead of FUSE and Python versus the overhead of our file system.

- `xmp` is the example file system which ships with fuse-python. We added support for extended attributes, using the same library, `py-xattr`, as used for TrueNames. Otherwise, it is a vanilla FUSE file system with no additional functionality.
- As a point of reference, we also include file system performance on a raw ext4 file system.

In the case of both TrueNames and `xmp`, we ran in single threaded mode, since `xmp` does not support multiple threads by default, and we wanted to modify it as little as possible. TrueNames will support multiple threads in the future. We disabled the inode cache, and set the `entry_timeout` to 0 in order to prevent stale file name entries. The only other modification to `xmp` was a fix for a bug which caused all writes to occur in append mode.

We ran two batches of experiments, one using an SSD, and one using a hard disk drive. Our SSD experiments were run on a 100 GB Intel 330 Series SSD, in an 8 core Intel Xeon CPU E3-1230 V2 @ 3.30GHz with 16 GB of RAM. Our HDD experiments were run on a 7200 RPM 500 GB Seagate Constellation drive, in an 8 core Intel Xeon CPU E5620 @ 2.40GHz and 24 GB of RAM. In both cases, we ran on Fedora with a 3.9.10-200.fc18.x86_64 kernel, and an ext4 file system as our backing store.

7.4 Results and Analysis

TrueNames is a proof of concept user space prototype, not a production file system, but even so, it has very low overhead, demonstrating that automatic naming can be added to production file systems with minimal performance implications. We tested TrueNames under a variety of micro benchmarks and macro benchmarks, in order to analyze its performance under extreme conditions as well as realistic workloads. TrueNames is noticeably slower under our micro benchmarks, as expected, but the performance penalty can be measured in fractions of a millisecond, and TrueNames exhibits no scaling issues under high load. Under normal file system loads, such as in our macro benchmarks, TrueNames performs with only a minimal overhead, much of it due to being Python and single-threaded. In addition, TrueNames shows no impact on operations other than file creation and extended metadata operations. In aggregate, these numbers show that an production implementation of these ideas can serve as an replacement for many other file systems, large and small alike, adding useful new functionality at little to no performance cost.

7.4.1 Microbenchmarks

There are a number of benchmarks designed to exercise metadata. However, these are generally aimed towards file system metadata, such as performing high-speed updates to modification times. Tools such as Filebench [2], while quite flexible, do not offer the ability to modify extended attributes out of the box. By contrast, we needed to evaluate our system's impact on extended metadata performance. In order to do this, we wrote a benchmark designed to add, update, and delete extended attributes continuously. In effect, if the file is of an automatically named type, this results in TrueNames continuously renaming the file.

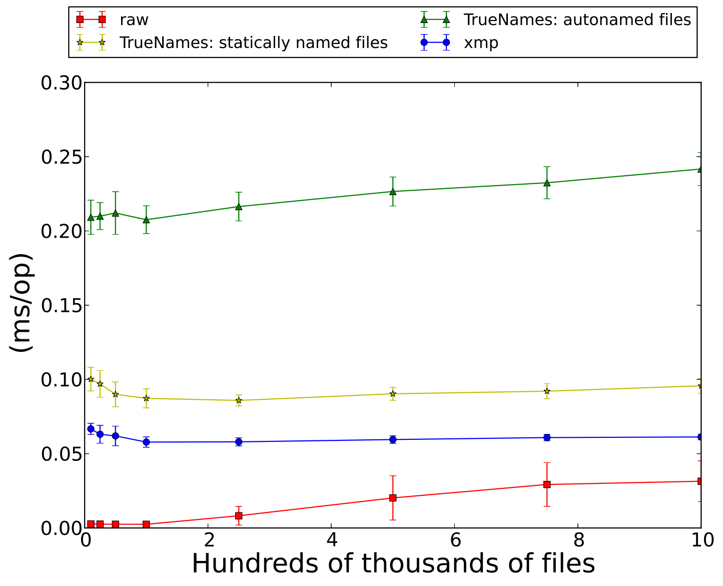
In order to focus as much as possible on the metadata speed, we pre-created a file set of size n. We then iterated over the file set until all n files had been touched, setting a single extended attribute on each file, calculating the latency of each operation and collecting statistics. Ext4 stores small metadata attributes in the inode, so files with a very large number of extended attributes will show lower performance than our benchmarks. One potential optimization is to ensure that metadata attributes relevant to the name are kept in the inode, since they are likely to be small, and not numerous.

Once the add benchmark completed, we ran similar benchmarks to update an attribute, and finally, to delete an attribute. This allowed us to exercise the new code paths continuously, highlighting any performance differences from our baseline file systems.

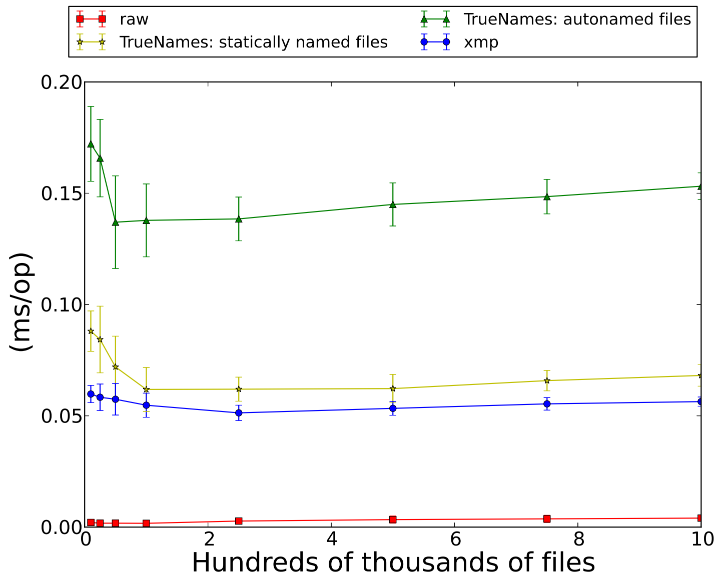
We ran this benchmark for both automatically named file types, and files which were not automatically named, in order to quantify the overhead incurred both with and without using the new features. We ran the metadata add, update and delete benchmarks for file sets from 10,000 to 1,000,000 files, which is comparable to modifying every file on a modern laptop at once. We then repeated each test forty times, in order to smooth noise and calculate a standard deviation.

Microbenchmarks are the most intensive tests, so it is unsurprising that they show the largest difference between the file systems. If we examine the difference between automatically named files and manually named files, we can see that automatic naming incurs a 100% penalty over metadata operations which do not affect the name, across add (Figure 7.2), update (Figure 7.3), and delete (Figure 7.3), and for both HDDs and SSDs. However, even at a 100% penalty, the additional cost can be measured in fractions of a millisecond. Looking at the difference between the baseline fuse system `xmp`, and TrueNames without name templates, we can see that there is approximately a 30% overhead to using TrueNames without automatic naming. This is primarily due to checking every time if the file has a template, which requires retrieving extended metadata, and therefore both an additional kernel crossing and potentially a disk access. In total, TrueNames adds four extra kernel crossings per one file creation. The additional kernel crossings are a performance issue specific to FUSE, and would not occur in an in-kernel file system.

Even at this high operation rate, and for a million files, we can see from the latencies that the disk cache is rarely saturated. This performance will occur in a small fraction of operations, usually during file creation, and the additional latency will be masked under most normal workloads, as we discuss in the next section. TrueNames shows a fixed overhead without scaling bottlenecks up to a million files, making it suitable for fairly large workloads.

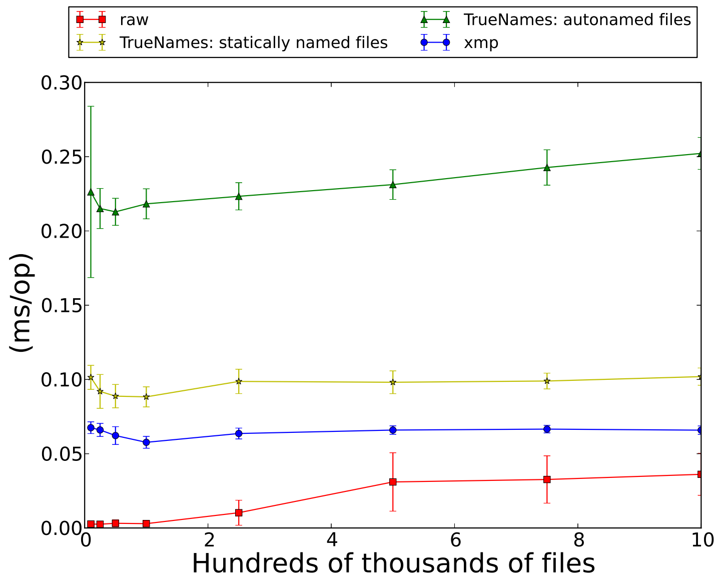


(a) HDD: Adding metadata

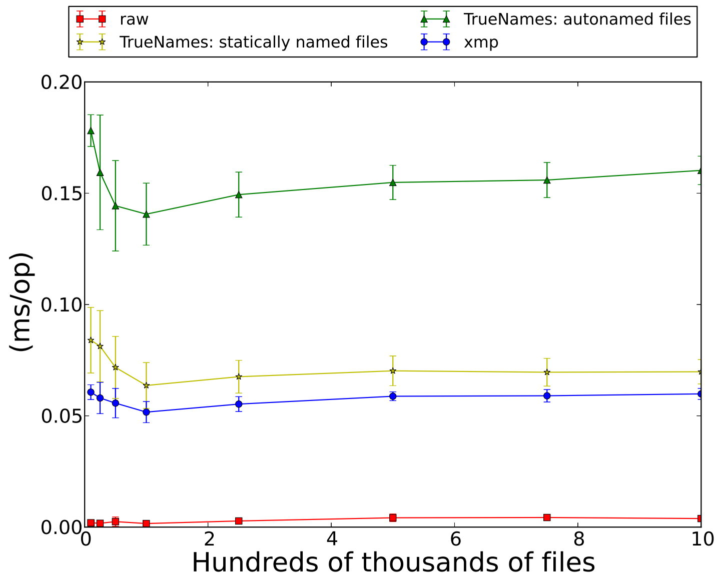


(b) SSD: Adding metadata

Figure 7.2: Latency and standard deviation per add extended metadata operation, for 40 runs of the benchmark. We tested TrueNames where the file either does or does not have an automatically named type, and on two baseline file systems, xmp and a raw ext4 system.

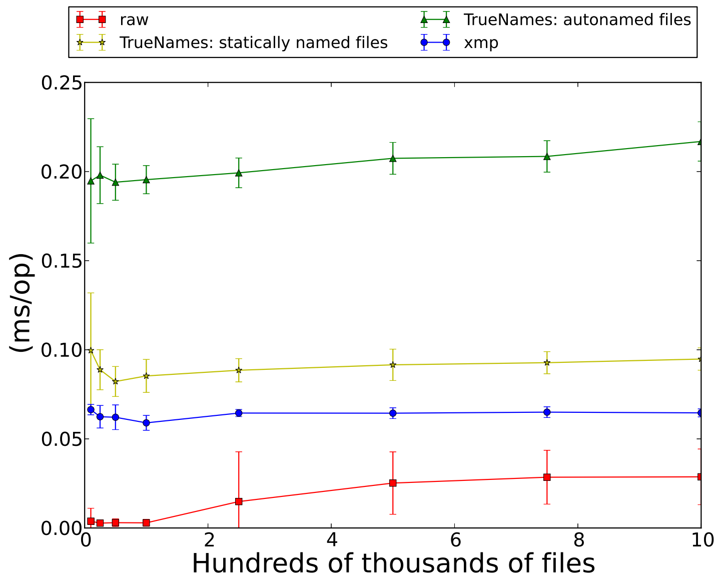


(a) HDD: Updating metadata

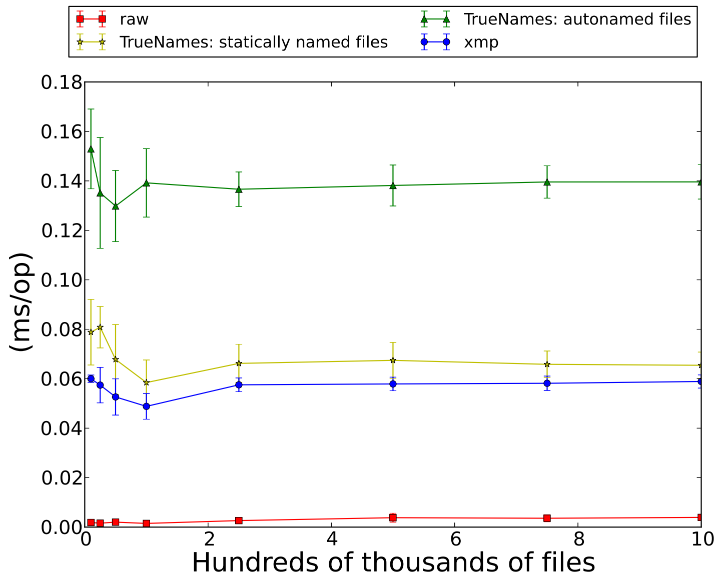


(b) SSD: Updating metadata

Figure 7.3: Latency and standard deviation per update extended metadata operation, for 40 runs of the benchmark. We tested TrueNames where the file either does or does not have an automatically named type, and on two baseline file systems, xmp and a raw ext4 system.



(a) HDD: Deleting metadata



(b) SSD: Deleting metadata

Figure 7.4: Latency and standard deviation per delete extended metadata operation, for 40 runs of the extended metadata operations benchmark. We tested TrueNames where the file either does or does not have an automatically named type, and on two baseline file systems, xmp and a raw ext4 system.

7.4.2 Macrobenchmarks

While micro benchmarks can be useful for setting an upper bound on performance, they are often an unrealistic assessment of how a file system will perform in practice. In the real world, file systems are experiencing a variety of operations from many different sources. In order to simulate the performance in a realistic environment, we chose a standard benchmark, Filebench [2]. This benchmark does not exercise the extended metadata functionality, and is therefore comparable to the statically named files experiments from section 7.4.1. We ran two different benchmark suites, the `fileserver` suite, which is designed to simulate the behavior of a typical file server, by performing a series of creates, deletes, appends, reads, writes and attribute operations on a directory tree. Mean directory size is 20 files, and the mean file size is 128kB. The workload generated is somewhat similar to SPECsfs [3]. We also ran the `createfiles` benchmark, which creates a specified number of files in a directory tree, with an average directory size of 100 files. File sizes are chosen according to a gamma distribution with a mean size of 16kB. We varied each of these from 1 to 32 threads, and from 100,000 to 1,000,000 files. We then repeated each test forty times, in order to smooth noise and generate a standard deviation.

For both the fileserver and createfiles benchmarks, TrueNames has file creation performance that is highly comparable with that of `xmp`, at about 15% overhead. File creation performance for the fileserver benchmark can be seen in Figure 7.5 for HDDs and 7.6 for SSDs. File creation performance for createfiles is in Figure 7.7 for HDDs and 7.8 for SSDs. Other typical operations, such as writing a file, deleting a file, or calling `stat` on a file, had insignificant overhead, meaning that TrueNames is suitable for all but the most create-intensive workloads. This demonstrates that most of the fixed overhead of TrueNames is masked by normal operation latencies.

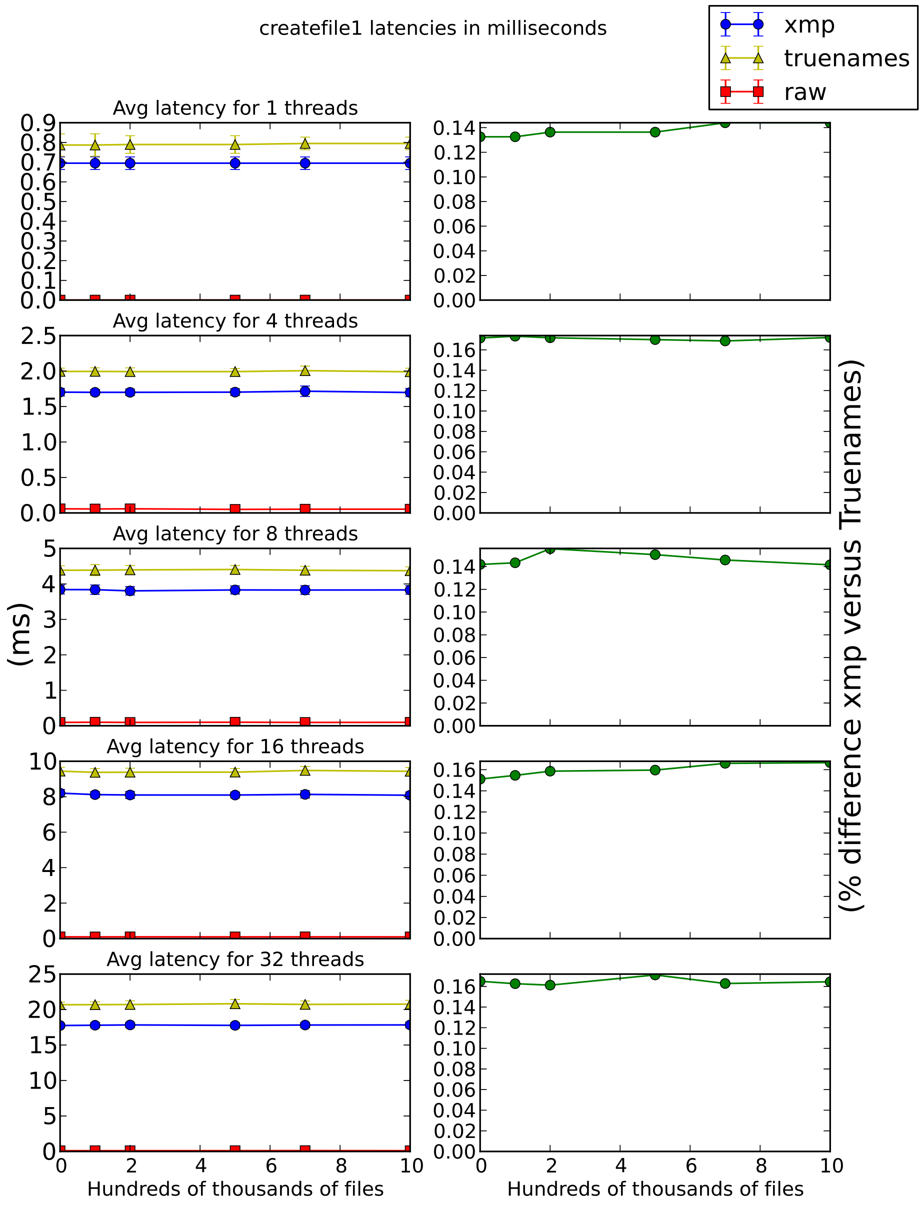


Figure 7.5: File creation times and standard deviation for 40 runs of the createfiles benchmark on a HDD, on TrueNames and on two baseline file systems, xmp and a raw ext4 system. We also show the percentage difference between xmp and TrueNames.

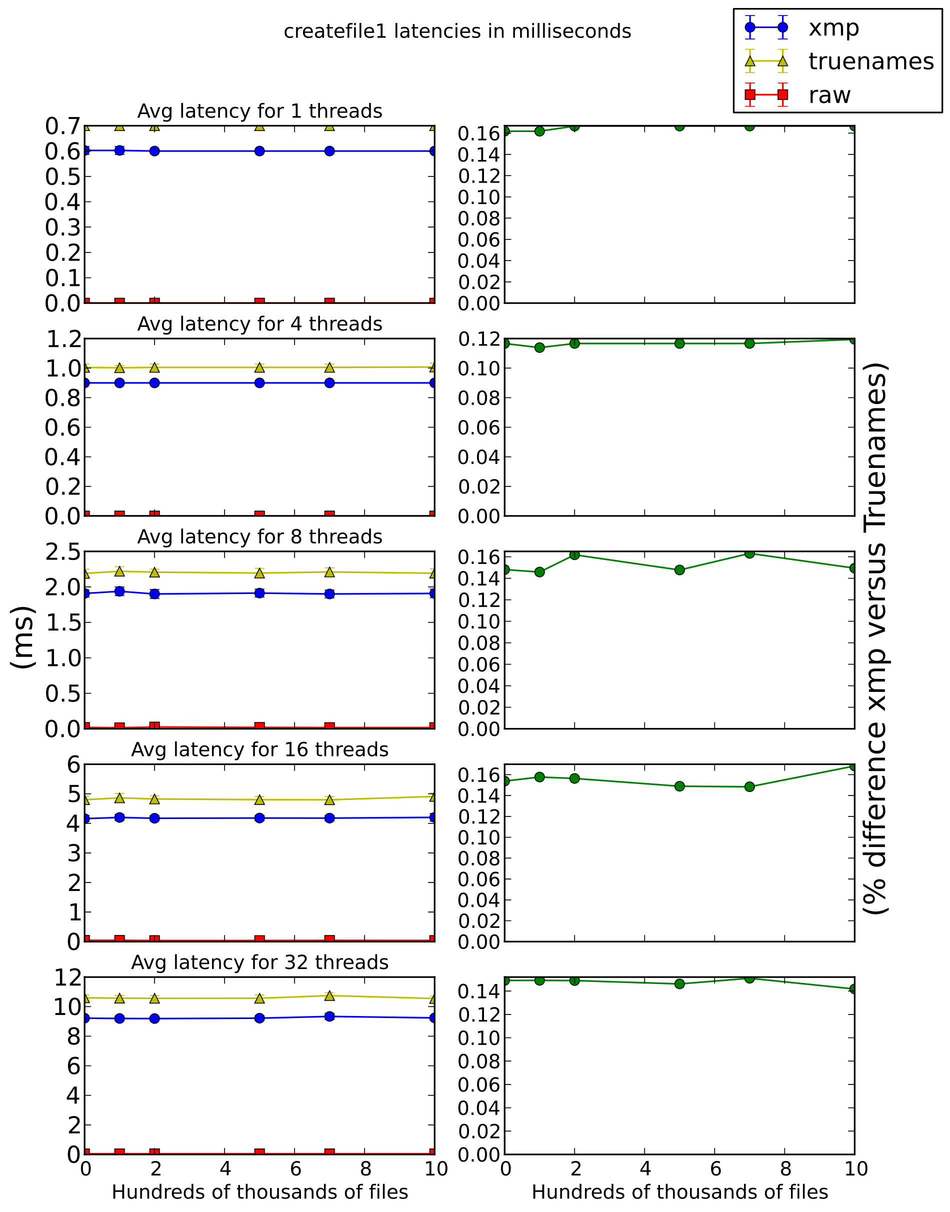


Figure 7.6: File creation times and standard deviation for 40 runs of the createfiles benchmark on an SSD, on TrueNames and on two baseline file systems, xmp and a raw ext4 system. We also show the percentage difference between xmp and TrueNames.

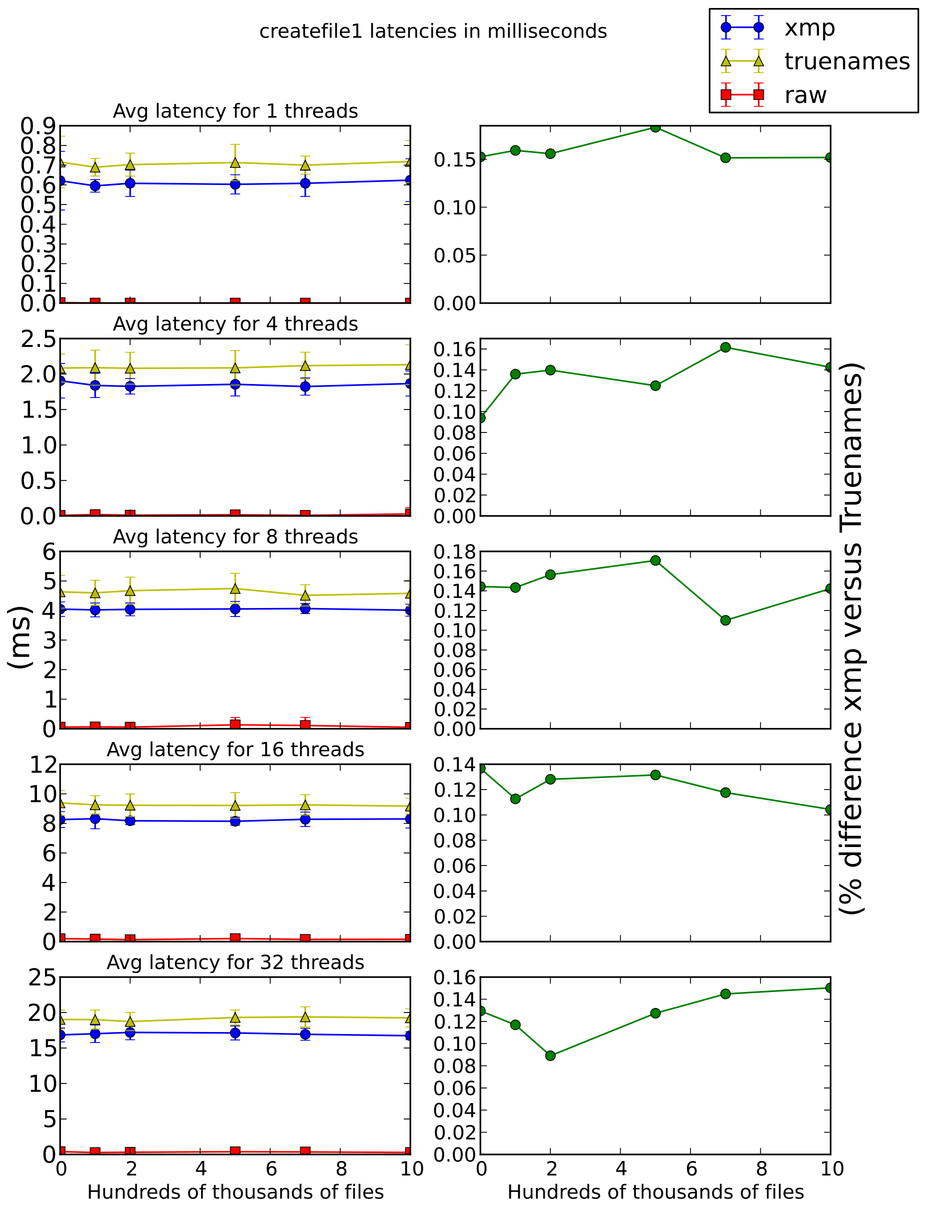


Figure 7.7: File creation times and standard deviation for 40 runs of the fileserver benchmark on an HDD, on TrueNames and on two baseline file systems, xmp and a raw ext4 system. We also show the percentage difference between xmp and TrueNames.

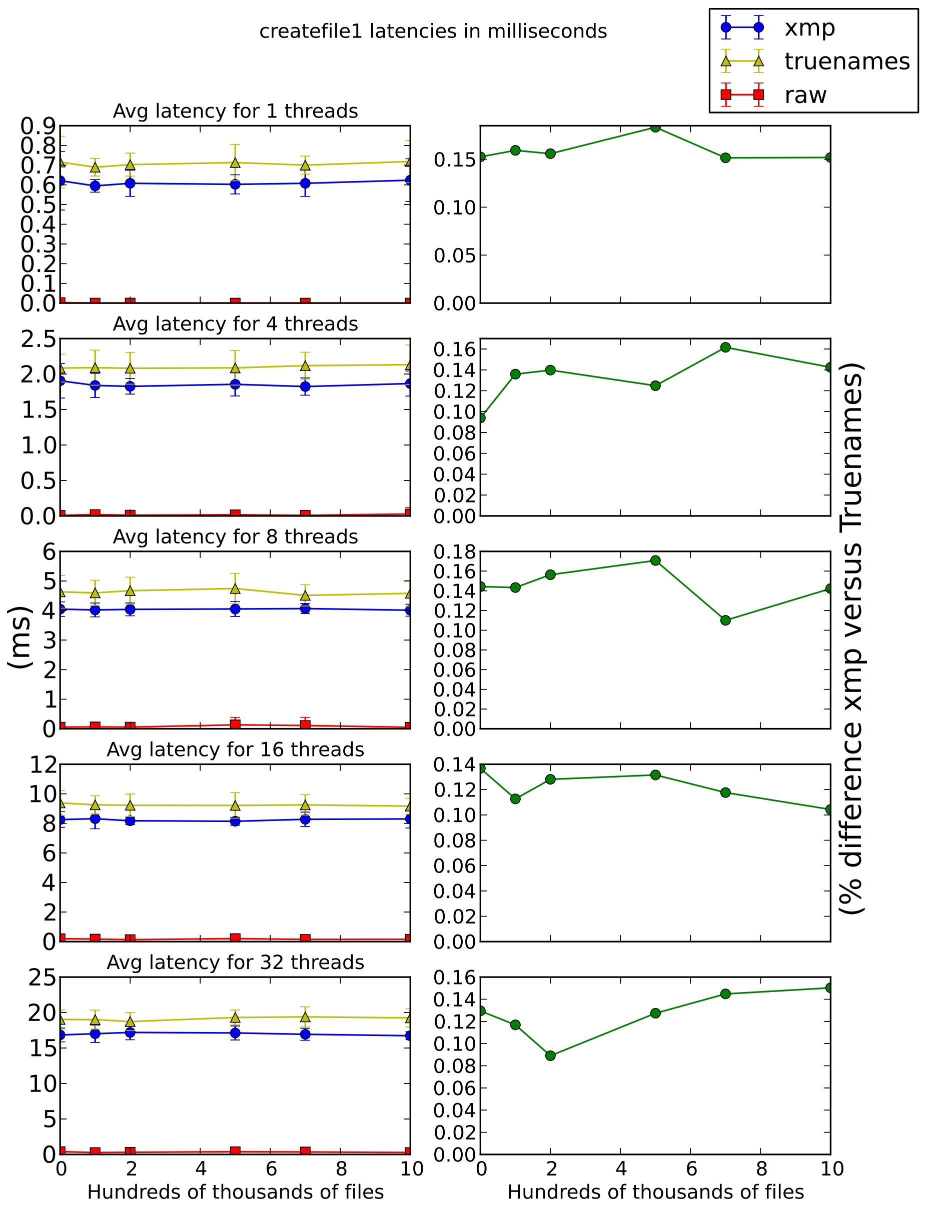


Figure 7.8: File creation times and standard deviation for 40 runs of the fileserver benchmark on an SSD, on TrueNames and on two baseline file systems, xmp and a raw ext4 system. We also show the percentage difference between xmp and TrueNames.

7.5 Conclusions

We have created a new method of file naming, *metadata aware file naming*, which adds a minimal 15% overhead under a variety of workloads, and scales up to millions of files. Metadata aware file naming provides a easy to use and efficient set of abstractions for managing file names and metadata. It eases file management by automating the process of file naming, and giving files truthful, structured, and automatically updated file names. It also gently encourages users and applications to supply more searchable metadata. Using metadata aware file naming can not only benefit users and application developers in the short term, it can ease the migration path to non-hierarchical file systems and improve search. By separating unique system identifiers from human readable file names, we enable file systems and users to work together to manage data more effectively.

Metadata aware file naming is broadly applicable, as we have shown by describing a variety of use cases in existing software which could be simplified by using our prototype filesystem, TrueNames. It can simplify application development, reduce data loss, and make it easier for users and applications to find and manage files. Additionally, we have found TrueNames to be an extremely useful tool during the writing of this work, and will be porting it to OS X in the near future in order to take advantage of it on more of our machines.

Chapter 8

Conclusions

The future is already here—it's just not very evenly distributed.

William Gibson

High end scientific computing is on the cutting edge of science and computing technology, and yet it is needlessly trapped in the past when it comes to file management. Three-ring binders, manual file organization and naming, and brute force search tools are inefficient for both the system and the user, and do not scale to modern data volumes. Without robust file management, we are all adrift on a sea of data. Ranked search will allow file systems to scale better, by reducing the number of results retrieved, and allowing the user to quickly refine searches. Automatic file naming allows the user to spend less time managing their files, application developers to spend less time in development, and enables users and applications to better work together at managing files.

Our work contributes the following research to the field of scientific file management. Scientific data management has not been a well understood field, and we have contributed to a better understanding of both scientific data practices, and scientific data. File naming is an uncharted area, and ours is the first research to focus on automatic file naming rather than directories. We have shown that automatically named files can be achieved at less than 15% overhead. File system provenance is a widely requested scientific tool, but one whose long-term properties are poorly understood. We have contributed a large body of data to the study of file system provenance, one which can be measured in years, rather than days, as well as providing new mechanisms for future studies of file system provenance. Finally, ours is the first application of provenance data in the area of file system ranking. We have demonstrated the feasibility of provenance based ranking, shown that it out-performs other ranking algorithms and described how it can improve both caching and file system search.

Our work is designed to make sense of the file system, managing data automatically and transparently, and freeing the file system from the vagaries of manually managed and searched files. Our algorithms are designed to be scalable, and easy to use and understand. However, there is still much work to do.

Provenance is an exciting new area of research, and we have just added a very large data set to the available resources. In the future, we intend to explore properties of provenance graphs over time, such as repeated structures, to see whether better provenance compression can be achieved. We also intend to explore using provenance for personalized file ranking, and pursue a full scale study for provenance ranking *in situ.*

TrueNames is still in the early stages of development, and we intend to explore whether further performance gains can be achieved. We also intend to evaluate TrueNames from the usability side, exploring how well it meets user needs.

Finally, we intend to explore how all of these advances can be woven together in the context of a non-hierarchical file system to achieve high quality scalable search.

File systems should work to the benefit of both the user and the computing system, and our research advances that goal, allowing scientists and systems to work together to manage and find files and metadata more effectively.

Bibliography

[1] Aperture. `http://www.apple.com/aperture/what-is.html`.

[2] Filebench. `http://sourceforge.net/projects/filebench/`.

[3] Filebench wiki: Pre-defined personalities. `http://sourceforge.net/apps/mediawiki/filebench/index.php?title=Pre-defined_personalities`.

[4] ID3 tag version 2.3.0. `http://id3.org/id3v2.3.0`.

[5] iPhoto. `http://www.apple.com/ilife/iphoto/`.

[6] Lightroom Help: The Filename Template Editor and Text Template Editor. `http://helpx.adobe.com/lightroom/help/filename-template-editor-text-template.html`.

[7] Mendeley Add & Organize. `http://www.mendeley.com/features/add-and-organize/`.

[8] MusicBrainz Picard. `https://musicbrainz.org/doc/MusicBrainz_Picard/Documentation`.

[9] The Open Archives Initiative Protocol for Metadata Harvesting. `http://www.openarchives.org/OAI/openarchivesprotocol.html`.

[10] Systems Research at Harvard: PASS Traces. `http://www.eecs.harvard.edu/syrah/pass/traces/`.

[11] Dryad. `http://www.datadryad.org/`, September 2012.

[12] HBase. `http://hbase.apache.org/`, September 2012.

[13] Metadata Encoding & Transmission Standard. `http://www.loc.gov/standards/mets/`, November 2012.

[14] Wide-field Infrared Survey Explorer (WISE) All-Sky Release. `http://irsadist.ipac.caltech.edu/wise-allsky/`, September 2012.

[15] Argo. `http://www.argodatamgt.org/`, March 2013.

[16] [fastbit-users] sparse data. `https://hpcrdm.lbl.gov/pipermail/fastbit-users/2012-October/001510.html`, Mar 2013.

[17] HDF5. `http://www.hdfgroup.org/HDF5/`, Jan 2013.

[18] E. Agichtein, E. Brill, and S. Dumais. Improving web search ranking by incorporating user behavior information. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '06*, page 19, New York, New York, USA, Aug. 2006. ACM Press.

[19] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, pages 31–45, Feb. 2007.

[20] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock. Kepler: An extensible system for design and execution of scientific workflows. 2004.

[21] A. Amer, D. D. E. Long, J.-F. Pâris, and R. C. Burns. File access prediction with adjustable accuracy. In *Proceedings of the 21st IEEE International Performance Conference on Computers and Cmmunication (IPCCC '02)*, Phoenix, Apr. 2002. IEEE.

[22] S. Ames, N. Bobb, K. M. Greenan, O. S. Hofmann, M. W. Storer, C. Maltzahn, E. L. Miller, and S. A. Brandt. LiFS: An attribute-rich file system for storage class memories. In *Proceedings of the 23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies*, College Park, MD, May 2006. IEEE.

[23] Apple Developer Connection. Working with Spotlight. `http://developer.apple.com/macosx/tiger/spotlight.html`, 2004.

[24] Apple Inc. iTunes. `http://www.apple.com/itunes/overview/`, Jan 2010.

[25] O. Bergman and R. Beyth-Marom. Improved search engines and navigation preference in personal information management. *ACM Transactions on Information Systems*, 2008.

[26] M. Bernstein and M. V. Kleek. Information scraps: How and why information eludes our personal information management tools. *ACM Transactions on Information Systems*, 2008.

[27] D. Bhagwat and N. Polyzotis. Searching a file system using inferred semantic links. In *Proceedings of the Sixteenth ACM Conference on Hypertext and Hypermedia*, HYPERTEXT '05, pages 85–87, New York, NY, USA, 2005. ACM.

[28] R. Boardman and M. A. Sasse. "Stuff Goes into the Computer and Doesn't Come Out": A Cross-tool Study of Personal Information Management. In *Proceedings of the 2004 Conference on Human Factors in Computing Systems (CHI '04)*, pages 583–590. ACM Press, 2004.

[29] S. Büttcher and C. L. A. Clarke. A document-centric approach to static index pruning in text retrieval systems. In *Proceedings of the 2006 International Conference on Information and Knowledge Management Systems (CIKM '06)*, pages 182–189, 2006.

[30] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. Vistrails: Visualization meets data management. June 2006.

[31] Z. Cao, T. Qin, T. Liu, M. Tsai, and H. Li. Learning to rank: from pairwise approach to listwise approach. *Proceedings of the 24th international conference on Machine Learning*, pages 129–136, 2007.

[32] D. Carmel, D. Cohen, R. Fagin, E. Farchi, M. Herscovici, Y. S. Maarek, and A. Soffer. Static index pruning for information retrieval systems. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '01)*, pages 43–50, 2001.

[33] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006.

[34] P. A. Chirita, R. Gavriloaie, S. Ghita, W. Nejdl, and R. Paiu. Activity based metadata for semantic desktop search. *Lecture Notes in Computer Science : The Semantic Web: Research and Applications*, pages 439–454, 2005.

[35] E. Chu, J. Beckmann, and J. Naughton. The case for a wide-table approach to manage sparse relational data sets. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 821–832, New York, NY, USA, 2007. ACM.

[36] S. Cohen and C. Domshlak. On Ranking Techniques for Desktop Search. *Communication*, pages 1183–1184, 2007.

[37] S. Dayal. Characterizing HEC storage systems at rest. Technical report, Carnegie-Mellon University, 2008.

[38] J. R. Douceur and W. J. Bolosky. A large-scale study of file-system contents. In *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '99, 1999.

[39] S. Dumais, E. Cutrell, J. Cadiz, G. Jancke, R. Sarin, and D. C. Robbins. Stuff I've seen: a system for personal information retrieval and re-use. In *Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 72–79. ACM Press, 2003.

[40] S. Fertig, E. Freeman, and D. Gelernter. Lifestreams: an alternative to the desktop metaphor. In *CHI '96: Conference Companion on Human Hactors in Computing Systems*, pages 410–411, 1996.

[41] J. Gaugaz, S. Costache, P.-A. Chirita, C. Firan, and W. Nejdl. Activity based links as a ranking factor in semantic desktop search. In *Proceedings of the 2008*

Latin American Web Conference, pages 49–57, Washington, DC, USA, 2008. IEEE Computer Society.

[42] D. Giampaolo. *Practical File System Design with the Be File Sstem*. Morgan Kaufmann, 1st edition, 1999.

[43] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole, Jr. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 16–25. ACM, Oct. 1991.

[44] B. Gopal and U. Manber. Integrating content-based access mechanisms with hierarchical file systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–278, Feb. 1999.

[45] H. Graven, A. Kozyr, and R. M. Key. Historical observations of oceanic radiocarbon conducted prior to GEOSECS. `http://cdiac.ornl.gov/ftp/oceans/Historical\_C14\_obs/`, 2012.

[46] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 197–207, 1994.

[47] T. Haveliwala and S. Kamvar. The second eigenvalue of the google matrix. *Stanford University Technical Report*, 2003.

[48] D. Hitz, J. Lau, and M. Malcom. File system design for an NFS file server appliance. In *Proceedings of the Winter 1994 USENIX Technical Conference*, pages 235–246, San Francisco, CA, Jan. 1994.

[49] A. Holloway. The purge threat: scientists' thoughts on peta-scale usability. In *Proceedings of the Sixth workshop on Parallel Data Storage*, PDSW '11, pages 31–36, New York, NY, USA, 2011. ACM.

[50] Y. Hua, H. Jiang, Y. Zhu, D. Feng, and L. Tian. SmartStore: A new metadata organization paradigm with semantic-awareness for next-generation file systems. In *Proceedings of SC09*, Nov. 2009.

[51] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. Pocock, P. Li, and T. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34(Web Server issue):729–732, July 2006.

[52] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST '04)*, pages 73–86, San Francisco, CA, Apr. 2004. USENIX.

[53] T. Joachims. Optimizing search engines using clickthrough data. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, pages 133–142, New York, NY, USA, 2002. ACM.

[54] K. S. Jones, S. Walker, and S. Robertson. A probabilistic model of information retrieval: development and comparative experiments: Part 1. *Information Processing & Management*, 36(6):779 – 808, 2000.

[55] K. S. Jones, S. Walker, and S. Robertson. A probabilistic model of information retrieval: development and comparative experiments: Part 2. *Information Processing & Management*, 36(6):809 – 840, 2000.

[56] S. Jones, C. Strong, D. D. E. Long, and E. L. Miller. Tracking emigrant data via transient provenance. In *Proceedings of the 3rd USENIX Workshop on the Theory and Practice of Provenance (TaPP '11)*, June 2011.

[57] S. Jones, C. Strong, A. Parker-Wood, A. Holloway, and D. D. E. Long. Easing the Burdens of HPC File Management. In *Proceedings of the 6th Parallel Data Storage Workshop (PDSW '11)*, Nov. 2011.

[58] W. Jones and A. Phuwanartnurak. Don't take my folders away!: organizing personal information to get things done. *CHI '05 Extended Abstracts on Human Factors in Computing Systems*, pages 1505–1508, 2005.

[59] V. Kalnikaité and S. Whittaker. Software or wetware?: discovering when and why

people use digital prosthetic memory. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 71–80, 2007.

[60] J. Kaye, J. Vertesi, S. Avery, and A. Dafoe. To have and to hold: exploring the personal archive. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 275–284, 2006.

[61] J. Koren, Y. Zhang, S. Ames, A. Leung, C. Maltzahn, and E. L. Miller. Searching and navigating petabyte scale file systems based on facets. In *Proceedings of the 2007 ACM Petascale Data Storage Workshop (PDSW '07)*, Reno, NV, November 2007.

[62] T. M. Kroeger and D. D. E. Long. Design and implementation of a predictive file prefetching algorithm. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 105–118, Boston, Jan. 2001.

[63] G. H. Kuenning. LASR Traces. `http://iotta.snia.org/traces/2`.

[64] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[65] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandier, L. Doshi, and C. Bear. The Vertica Analytic Database: C-store 7 Years Later. In *Proceedings of the 38th Conference on Very Large Databases (VLDB '12)*, volume 5, pages 1790–1801, 2012.

[66] N. Lester, A. Moffat, and J. Zobel. Fast on-line index construction by geometric partitioning. In *Proceedings of the 2005 International Conference on Information and Knowledge Management Systems (CIKM '05)*, Nov. 2005.

[67] A. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller. Spyglass: Fast, scalable metadata search for large-scale storage systems. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*, pages 153–166, Feb. 2009.

[68] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the 2008 USENIX Annual Technical Conference*, June 2008.

[69] T. W. Malone. How do people organize their desks?: Implications for the design of office information systems. *ACM Transactions on Information Systems*, 1(1):99–112, Jan. 1983.

[70] U. Manber and S. Wu. Glimpse: A tool to search through entire file systems. Technical Report TR 93-34, The University of Arizona, Oct. 1993.

[71] D. Margo, P. Macko, and M. Seltzer. Addressing underspecified lineage queries on provenance. Technical report, Harvard University, 2012.

[72] S. Mitra, M. Winslett, and W. W. Hsu. Query-based partitioning of documents and indexes for information lifecycle management. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, June 2008.

[73] K.-K. Muniswamy-Reddy and D. A. Holland. Causality-based versioning. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2009.

[74] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference*, Boston, MA, 2006.

[75] J. Naps, M. Mokbel, and D. Du. Pantheon: Exascale file system search for scientific computing. In J. Bayard Cushing, J. French, and S. Bowers, editors, *Scientific and Statistical Database Management*, volume 6809 of *Lecture Notes in Computer Science*, pages 461–469. Springer Berlin / Heidelberg, 2011.

[76] M. A. Olson. The design and implementation of the Inversion file system. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 205–217, San Diego, California, USA, Jan. 1993.

[77] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[78] Y. Padioleau and O. Ridoux. A logic file system. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 99–112, San Antonio, TX, June 2003.

[79] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford, Nov. 1998.

[80] A. Parker-Wood, D. D. E. Long, E. L. Miller, M. Seltzer, and D. Tunkelang. Making sense of file systems through provenance and rich metadata. Technical Report UCSC-SSRC-12-01, University of California, Santa Cruz, Mar. 2012.

[81] A. Parker-Wood, B. Madden, M. McThrow, D. D. E. Long, I. Adams, and A. Wildani. Examining extended and scientific metadata for scalable index designs. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR 2013)*, June 2013.

[82] S. Patil and G. Gibson. Scale and concurrency of GIGA+: File system directories with millions of files. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2011.

[83] S. Patil, G. A. Gibson, G. R. Ganger, J. Lopez, M. Polte, W. Tantisiroj, and L. Xiao. In search of an API for scalable file systems: under the table or above it? In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[84] S. V. Patil, G. A. Gibson, S. Lang, and M. Polte. GIGA+: scalable directories for shared file systems. In *Proceedings of PDSW '07*, 2007.

[85] S. Raub, A. Dingle, and D. Stanton. "Temporal ranking scheme for desktop searching", U.S. Patent 7529739, May 2009.

[86] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network file system. In *Proceedings of the Summer 1985 USENIX Technical Conference*, pages 119–130, 1985.

[87] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the Conference on File and Storage Technologies (FAST)*, pages 231–244, Jan. 2002.

[88] S. Shah, C. A. N. Soules, G. R. Ganger, and B. D. Noble. Using provenance to aid in personal file search. In *Proceedings of the 2007 USENIX Annual Technical Conference*, pages 171–184, June 2007.

[89] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS version 4 protocol. IETF Network Working Group RFC 3010, Dec. 2000.

[90] C. A. N. Soules and G. R. Ganger. Connections: using context to enhance file search. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 119–132, New York, NY, USA, 2005. ACM Press.

[91] C. Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904.

[92] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: A column oriented DBMS. In *Proceedings of the 31st Conference on Very Large Databases (VLDB)*, pages 553–564, Trondheim, Norway, 2005.

[93] C. Strong, S. Jones, A. Parker-Wood, A. Holloway, and D. D. E. Long. Los Alamos National Laboratory Interviews. Technical Report UCSC-SSRC-11-06, University of California, Santa Cruz, Sept. 2011.

[94] D. A. Talbert and D. Fisher. An empirical analysis of techniques for constructing and searching k-dimensional trees. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '00, pages 26–33, New York, NY, USA, 2000. ACM.

[95] S. K. Tyler and J. Teevan. Large scale query log analysis of re-finding. In *Proceedings of the third ACM international conference on Web search and data mining*, WSDM '10, pages 191–200, New York, NY, USA, 2010. ACM.

[96] A. Vahdat and T. Anderson. Transparent result caching. In *Proceedings of the USENIX Annual Technical Conference*, number 98, New Orleans, LA, 1998.

[97] R. van Heuven van Staereling, R. Appuswamy, D. van Moolenbroek, and A. Tanenbaum. Efficient, Modular Metadata Management with Loris. In *Networking, Architecture and Storage (NAS), 2011 6th IEEE International Conference on*, pages 278 –287, July 2011.

[98] R. van Zwol, L. Garcia Pueyo, M. Muralidharan, and B. Sigurbjörnsson. Machine learned ranking of entity facets. In *Proceedings of the 33rd international ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '10, pages 879–880, New York, NY, USA, 2010. ACM.

[99] E. M. Voorhees. Variations in relevance judgments and the measurement of retrieval effectiveness. *Information processing & management*, 36(5):697–716, 2000.

[100] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, D. D. E. Long, and T. T. McLarty. File system workload analysis for large scale scientific computing applications. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 139–152, College Park, MD, Apr. 2004.

[101] R. W. Watson and R. A. Coyne. The parallel I/O architecture of the High Performance Storage System (HPSS)). In *Proceedings of the 14th IEEE Symposium on Mass Storage Systems*, pages 27–44, Monterey, CA, Sept. 1995.

[102] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the Panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, pages 17–33, Feb. 2008.

[103] S. Whittaker. Personal Information Management: From Information Consumption to Curation. *Annual Review of Information Science and Technology*, pages 1–103, 2011.

[104] G. A. S. Whittle, J.-F. Pâris, A. Amer, D. D. E. Long, and R. Burns. Using multiple predictors to improve the accuracy of file access predictions. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 230–240, Apr. 2003.

[105] K. Wu, K. Stockinger, and A. Shoshani. Breaking the Curse of Cardinality on Bitmap Indexes. *Scientific and Statistical Database Management*, pages 1–15, 2008.

[106] Y. Xie, K.-K. Muniswamy-Reddy, D. D. E. Long, A. Amer, D. Feng, and Z. Tan. Compressing provenance graphs. In *Proceedings of the 3rd USENIX Workshop on the Theory and Practice of Provenance (TaPP '11)*, June 2011.

[107] Z. Xu, M. Karlsson, C. Tang, and C. Karamanolis. Towards a semantic-aware file store. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*, May 2003.

[108] T. Yeh, D. D. E. Long, and S. A. Brandt. Using program and user information to improve file prediction performance. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS '01)*, pages 111–119, Tucson, Nov. 2001. IEEE.

[109] L. Zhang and Y. Zhang. Interactive retrieval based on faceted feedback. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '10, pages 363–370, New York, NY, USA, 2010. ACM.

[110] M. Zhu, S. Shi, N. Yu, and J.-R. Wen. Can phrase indexing help to process non-phrase queries? *Proceeding of the 17th ACM conference on Information and knowledge mining - CIKM '08*, page 679, 2008.