UNIVERSITY OF CALIFORNIA
SANTA CRUZ

THE PROMISE OF DATA GROUPING IN LARGE SCALE STORAGE SYSTEMS

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

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Avani Wildani

September 2013

The Dissertation of Avani Wildani
is approved:

Professor Ethan Miller, Chair

Professor Darrell Long

Professor Ahmed Amer

Jiri Schindler, Ph.D.

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

The Promise of Data Grouping in Large Scale Storage Systems

by

Avani Wildani

"Big Data" is one of the most prominent buzzwords of our age, and for good reason. From genetics to supercomputers to Facebook, big data is everywhere, and managing this data is going to be essential in the years to come.

One of the most basic interactions between a user and a storage system is the ability to retrieve data quickly and reliably. My thesis centers around the use of statistical properties of I/O traces to identify sets of data that are accessed together in order to store big data reliably and efficiently. We begin by showing that data can be assigned to predictive groups scalably with minimal domain knowledge and system impact. We then show how these groupings spur a variety of substantial improvements in areas including power management, performance, reliability, and resource allocation.

To Ran, for telling me not to go to grad school,

To the people of Harvey Mudd,

To Chris, Melissa, Shamik, and the rest of the family that kept me sane,

To Peter, for too much to list,

And, most importantly, to my father, the original Dr. Gadani.

Acknowledgments

I would like to thank my advisor, Prof. Ethan Miller, for exploring ideas with me and sharing his very valuable insights, my co-authors Thomas Schwarz, Darrell Long, Lee Ward, Ohad Rodeh, and Ian Adams for their multiple contributions to the work presented in this thesis, the rest of my committee for their feedback, and the Storage Systems Research Center, for the chocolate, whiskey, distractions, and red ink.

Chapter 1

Introduction

> *The ending is nearer than you think, and it is already written. All that we have left to choose is the correct moment to begin.*
>
> **– Alan Moore**

Computer science is the study of the layers of abstraction in how humans interact with machines, both physical and mathematical constructions. In computer systems, we focus this study to those abstractions that affect how users store, organize, and retrieve information using computational hardware. A pressing issue in systems is management of "big data," data that is too massive to immediately process and which leads to results that themselves must be processed and stored. Moreover, this big data is likely to be stored on a petascale or exascale storage system that is designed around the paradigm of "cloud computing," meaning that the multi-user, multi-application system must appear as a dedicated server to each user.

This thesis demonstrates the benefit of laying out data according to predicted access patterns for improved power management, data reduction, and fault isolation. Our statistical and machine learning based grouping algorithms answer questions such as "what can a grouping be based on?" and "is a given grouping meaningful for a given application?" We design our models to be flexible and require minimal workload-specific information so that our results are as broadly applicable as possible. We intend for this work to provide a launchpad for future specialized system design using groupings in

combination with caching policies and architectural distinctions such as tiered storage to create the next generation of scalable storage systems.

We define a *group* as any set of blocks, files, or objects that are likely to be accessed together. We also introduce a stronger notion of a *clique-group*, which adds the constraint that every data point is highly likely to be accessed within a short time of every other data point in the set. Groups typically arise from working sets of user or application data, though they can also represent higher level correlations such as inter-application dependencies and operating system interactions.

While individuals stored small amounts of data on separate individually managed storage devices, prior work showed little benefit to identifying patterns in data beyond a simple "hot or cold" model for caching. Most of the work that does use data access or content relies on domain knowledge or difficult to maintain metadata. Grouping data for access locality has its roots in the beginnings of modern computing; the Berkeley Fast File System [65] arranged blocks of data with their corresponding inodes on the same cylinders to limit disk head movement in 1984. Since then, the main focus of locality-aware storage has shifted to keeping popular data easy to access and away from keeping related data co-located.

Locating data based on popularity only works if data has predictable cycles of "boom and bust"; *i.e.*, if a piece of data once popular is likely to stay popular long enough to justify the cost of moving the data to the popular area and offset the loss of the longer spatio-temporal locality that we focus on. We are increasingly seeing workloads where a very small subset of data has high probability of being accessed, and a larger but dynamic subset of data has a high co-access probability. This type of workload arises in storage systems with multiple users running multiple applications, each of which brings in a different working set of data. While no particular file or block has a globally high probability of access, over a small window files are likely to be accessed if the working set they are part of has been accessed recently. Modern computers have the spare processing power to be able to find these correlations in large enough windows to be predictive and useful.

Popularity-based grouping data within a disk has been shown to help avoid track boundaries [93], isolate faults within a disk [100], and avoid power consumption from excessive disk activity and disk spin-ups [80]. On exascale systems, these benefits are magnified because every element in a group may be on separate disks, necessitating many extra spin-ups that both waste power and decrease the lifetime of the system [121, 81]. We used these studies to choose what applications to test our spatio-temporal groupings on multi-user, multi-purpose storage environments.

The rest of this work is divided into two parts, answering the questions "how do we find groups?" and "for what purposes are groups useful?" Chapter 4 discusses our classification approach to group identification using both categorical features in metadata and file content and statistical features derived from file and block level access patterns.

Grouping based on categorical features such as file contents or metadata is more likely to closely track semantic relationships between data elements, but the classification relies on collecting rich metadata, processing actual data content, or relying on an administrator to correctly and consistently label data to indicate group membership. Maintaining this type of categorization is incredibly difficult: files can be renamed, losing associated metadata, or a labeling scheme can change, losing relationships between old data and new writes. Moreover, the computational overhead of collecting and analyzing this type of data is prohibitive in systems that rely on fast disk I/O.

We show that with the level of input or output requests per second (IOPS) in modern storage systems, it is possible to collect enough data in real time to identify statistically based groupings that can predict future access patterns. Processing this data quickly is a challenge that we meet by using minimal data and "loose" classification. We discuss a variety of techniques including graph theoretic clique formation and agglomerative clustering, but we primarily focus on a naïve statistical method we developed called N-Neighborhood Partitioning (NNP). NNP runs in $O(n)$ and is less memory intensive than our other methods, making it the best choice for a high I/O rate environment with quickly shifting workloads. Slower techniques are reserved for more steady workloads where calculations can be done over a long period of time and data can be re-arranged

lazily.

To answer our second question, we examine applications of grouping to pressing issues in scalable storage. There are essentially two classes of grouping applications, those that use groups to inform data layout and those that use groups to prefetch data into a limited size cache. We use groups to optimize layout for power management and fault isolation, and we use groups to prefetch data into cache for data reduction through primary deduplication.

Power is a primary concern for both current and future datacenters. As hardware costs have dropped, power costs have risen. In addition, the hard drive is one of the primary factors that contributes to data center heating, making it a leading driver of cooling costs[84]. While some modern data centers leverage virtualization to reduce their hardware requirements by allowing more hardware to be shared, they must inevitably upgrade their hardware to meet the flood of data that future systems will be expected to store. The overall data center power cost restricts the size of data centers and where the data centers can be reasonably placed. On the scale of single disks, there has been work predicting idle periods where a single disk can be spun down [17]. PARAID [119] similarly powers down multiple disks during periods of light system load. Chapter 5 discusses our work to adapt grouping techniques to power management on large systems composed of many disks. In this scenario, power is saved by keeping disks idle and loading an entire working set – which is stored contiguously but may be stored on several disks – into cache when a member of the working set is requested. Our results show that the potential for power savings is very high in archival systems with selective spin-down such as Pergamum [108] in exchange for a minimal, periodic layout cost.

Another key concern for a large scale storage system is data availability. Rebuild time is a function of the I/O speed of the components of the system, *e.g.* disks. As disks increase in size, the time to read the entire disk, and thus the rebuild time, correspondingly increases. Exacerbating this problem, many larger archival systems sacrifice rebuild speed in exchange for increased reliability. These systems typically place data evenly across disks in order to distribute and thus dilute the impact of data loss. However,

failure events that do not lead to data loss are much more common, and this distribution ensures that the rebuild impacts a wide array of people or projects even though each could have only a very small amount of data in the restore queue. By combining existing reliability measures with selectively laying out data by working set, we show in Chapter 6 that we can use automatically derived groups to isolate failures such that they impact fewer users or projects, increasing the perceived availability of the system.

Ideally, failures in a system with group-based layout will fall to projects that see no accesses during the rebuild. However, even if the failure affects an active project, the difference in rebuild time for rebuilding one data block or rebuilding several on the same disk is fairly constant [124]. The probability of massive data loss is higher when an entire project is on the same device, and we mitigate this by suggesting additional parity or tertiary replicas. We also pre-fetch projects into a read cache to prevent repeat accesses in close succession to the same disk. We show that with restricted group sizes, we can save thousands of group hours over a three year trace for even an archival workload.

Data reduction, whether through compression or deduplication, is another major concern for large scale storage. Currently, the amount of data humans produce electronically is growing faster than the amount storage we build, resulting in necessary information deletion. Much of this data, however, are actually redundant copies. For example, a company storage server may store several copies of an e-mail sent to a project team, including any attachments. Many companies are even replacing traditional stand-alone development environments with banks of virtual machines stored on a small group of storage servers. These systems have high degrees of data duplication even though they are not storing backups. In-line deduplication for primary workloads is limited by the necessity of fast I/O, which leads to an unaffordably large deduplication index in memory to check for duplicate chunks. Given typical memory to disk cost ratios, deduplication becomes financially unsound with even conservative block sizes if the entire index must have fast access. Thus, it is not possible to keep the entire deduplication index in memory, which leads to frequent resource intensive disk accesses. This is known as the *disk bottleneck problem*.

In Chapter 7, we discuss how we designed a framework for deduplication index management that arranges fingerprints in the on-disk index cache according to data group membership. It then pre-fetches entire groups of fingerprints into the memory cache, achieving most of the benefit of an in-memory cache at a fraction of the cost. We show that we can deduplicate 80% of duplicate disk accesses with just 1% of the index cache size. This is significantly better that the basic chunk popularity models currently in use in production systems.

Grouping is likely to be an essential component of how future scalable and cloud systems are designed, and we have introduced a fast, accurate, lightweight grouping method and shown how it significantly improves storage of archival, educational, and research workloads. In Chapter 6.3 we discuss how our techniques could impact other workload types, particularly HPC and virtual machines. We also posit how different groupings and changes in group membership over time can be used to predict the type of workload. Finally, we discuss future areas where we believe the grouping methodologies in this thesis can be applied, as well as how to optimize hardware configurations based on the expected workload and working sets.

Chapter 2

Background

Systems programmers are the high priests of a low cult.

– Bob Barton

The goal of data placement research is to assign data to physical and logical locations to balance the demands of accessing data efficiently, reliably, and correctly while minimizing the strain on the underlying storage system. From the early days of UNIX, storage researchers have designed systems that leverage the use and placement of data. One of the earliest examples is the Berkeley Fast File System, where data blocks were placed together with their inodes in the same cylinder groups to help prevent rapid arm movements across sectors of the disk during a read [65]. Modern systems have extended this paradigm of hardware-aware data placement, often abstracted to provide locality on the same server or same rack.

While it is trivial to place data with its metadata, it is much more difficult and interesting to place data adjacent to other data it is likely to be accessed with. The interest in the problem comes from trying to predict the next access (or the composition of the next n accesses) with as little information as possible; there is no restriction on what the next access could be, so the problem is completely probabilistic. When related data is scattered across a storage system, accessing the entire data group requires looking in multiple locations, increasing the response time and power consumption of the access while sometimes decreasing the reliability of multiple devices. We refer to this issue as

group fragmentation.

Though there has been a large amount of work in grouping for immediate prediction, answering the question "what are we likely to see next", this is not quite the problem we are interested in. Instead, this work helps answer the question of "given that x is accessed, what is likely to be accessed soon afterwards?". There is a subtle difference in the question formulation that leads to a very different set of grouping designs. For example, we disregard popularity of blocks, since our goal is to identify what is accessed together, whether or not any elements are likely to be accessed at all. This allows us to make predictions about the long tail of accesses that methods based on popularity cannot do. We do, however, use popularity at the group level to determine what groups are more relevant to the current workload. Since we are looking for patterns that represent long term trends, we also do not need to calculate groups in real time. We can instead form groupings that have more predictive capacity by aggregating longer traces and calculating groups in the background. Because these groupings are determined from more data, we can form useful groups with minimal metadata, which Riska *et al.* have shown can be collected with very low system overhead with a protocol analyzer [88]. This allows us to extend the benefits of our grouping methods and resulting frameworks to high performance workloads.

In the remainder of this section, we discuss different approaches researchers have taken for data grouping in various environments along with how they contrast with and complement this work. We also look at grouping work that was done for specific application areas and discuss how our generalized grouping techniques address the needs of these applications. We then briefly discuss the types of workloads best suited to a grouping based layout. Finally, we look at the similar problems of caching and tiered storage and compare the approaches we take to work in these fields.

2.1 Grouping

Grouping data according to use or projected use is an intuitive method of organizing a storage system. When a file is manually placed into a directory in a hierarchical file system, the file is being placed into a group in order to reduce the time to locate the file for a program or a user who finds files by traversing the directory tree. From a systems perspective, data was first grouped to speed up the time of access, such as the original BSD FFS which aimed to localize associated data and metadata blocks within the same cylinder groups to avoid excessive seeks [65].

For this work, we are more interested in identifying groups of elements that are likely to be accessed together over the long term. As a result, we do not value data popularity or recency of access as separate metrics when constructing our groupings, though we do use similarity between access times, which is a related idea. Identifying these groups has been a hot research area for some time, with significant work done to bridge the gap between the semantic information provided by the file system and the behavior of the physical disk controller [6, 101, 100]. Early multiprogramming systems used spatial and limited temporal locality to allocate data within pages, though they assumed a Gaussian group size distribution that we do not find in our workloads [16]. This body of previous work is a springboard, but cannot entirely solve the large scale, multi-user, multi-application grouping problems we are presented with. For example, we must be able to identify interleaved groups with minimal metadata, which current work does not address. Additionally, many systems we work with do not have extra bandwidth for trace collection: we must rely on grouping using only data location and timestamp. We specifically choose these two parameters because we want our work to generalize to as many real systems as possible, and every workload we have encountered thus far has contained this spatio-temporal data.

Arpaci-Dusseau *et al.* have made a variety of advances in semantically aware disk systems [101, 6]. Their work shows that if domain specific metadata is available, determining working sets using this metadata is significantly more reliable than domain agnostic approaches. If the metadata is unavailable, as is the case for many deployed sys-

tems with high performance expectations, we believe that domain-blind methods may prove sufficient for some workloads [123]. In Section 4.1 we discuss our method for determining working sets from workloads without accompanying metadata.

Early attempts at grouping detection used information retrieval techniques such as frequent sequence mining. Frequent sequence mining involves tracking portions of text that appear frequently in a trace and creating a database. One example project, C-Miner, uses frequent sequence matching on block I/O data, using a global frequent sequence database [61]. Frequent sequence matching is vulnerable to interlaced working sets within data and thus best for more specialized workloads, whereas the grouping techniques we will discuss in Chapter 4 are suitable for multi-application systems. Additionally, information retrieval techniques tag data with many metadata elements to facilitate search. This creates massive indices that create their own access problems, as the metadata space is huge and sparse [78].

Essary and Amer provide a strong theoretical framework for dynamically grouping blocks nearby together on a disk [33]. Amer's predictive methods have shown good results by offering the choice of "no prediction," allowing a predictor to signal uncertainty in the prediction [3]. Their work provides the theoretical optimal bounds for data grouping and is a primary foundation of our work. We extend this work by providing a realistic prediction mechanism and semi-permanent groupings, reducing the need for constant prediction. Stable access patterns are what we rely on for group detection: groups are found by looking for repeat correlated accesses. Paris *et al.* show that files in most workloads have stable access patterns [77].

Dorimani *et al.* demonstrate file-level grouping using static, pre-labeled groups, where the mean group size in blocks is about an order of magnitude larger than the mean file size [27]. Pre-labeled groupings such as these are hard to obtain for general workloads, and they are susceptible to evolving usage patterns and other variation in workload. By focusing on the core issue of inter-access similarity, we hope to be able to form dynamic groups from real-time access data instead of relying on domain knowledge.

Oly and Reed present a Markov model to predict I/O requests in scientific appli-

cations [76]. By focusing on scientific applications, their work bypasses the issue of interleaved groups that affect almost all automated grouping since their workloads are serialized experiments that. However, there is no indication that their results will hold for future applications even in the scientific domain. Grouping must be done in a way that is as domain agnostic, and thus as future proof as possible.

2.2 Workloads

We designed our grouping techniques to be workload agnostic, but there are certain types of workloads that are particularly good fits for grouping based layout. A workload should either need fast calculation and have high IOPS, leading to accurate but ephemeral groupings, or have low IOPS but need recalculation less quickly. In these systems, it is possible to find groups that retain predictive power over days or weeks.

2.2.1 Archival-by-Accident Workloads

One environment that contains cold data for many different use cases is commonly known as "archival" in the literature. Traditionally, data layout on archival systems has been inconsequential outside of keeping a small cache for popular data. However, archival systems are rapidly moving beyond the realm of corporate filing cabinets and into the realm of personal, and hence cultural, memory banks. We are seeing the first generations that record their entire lives and expect this data to be digitally stored perpetually and inexpensively [70, 114]. According to Gantz *et al.*, 281 exabytes of digital information was created in 2007, and they expect this number to grow tenfold by 2011 [36]. While the majority of this personal data begins life as data in active use, over time people lose interest in vacation photos and scanned receipts and this data ends up unconsciously filed away yet living among actively used data. Although this data is never explicitly archived, accesses to this data gradually start to resemble an archival workload.

Though companies such as Facebook are beginning to identify stale data and place it

in a different storage tier, current techniques are either very coarse or rely on expensive to obtain and maintain domain knowledge about how the data is used. For example, Facebook has significant amounts of metadata about each image in their store that they likely use for co-placement. We look, instead, at a more general case where the overhead to maintain metadata is prohibitive due to the variety of data on the storage system. We categorize workloads that gradually drift towards archival accesses as *archival-by-accident*.

A pressing example of an archival-by-accident system is the World Wide Web. Recent studies have shown that the top 10 websites account for 40% of web accesses, and the drop off is exponential instead of long tail, indicating that there are strong working sets in any index such as the Internet archive that stores web data [68]. We also know that typical storage systems cannot cope with the scale of archival data [8]. Handling these heterogeneous and archival-by-accident systems is going to be the next challenge for designers of archival systems.

The largest archival-by-accident system is currently archives of the world wide web. Media sharing sites such as Flickr™and YouTube™also exhibit this quality in their workloads. The prevalence of archival-by-accident systems is a strong contemporary motivation for addressing this problem at a broader scale than previous work in hierarchical storage management [38].

2.3 Data Layout Applications

Our primary aim for researching data grouping is to address issues that occur from rapidly expanding data sizes and storage systems. There are several big questions that the storage community needs to answer, and that data grouping can assist with;

- How can we keep the power costs in check as a storage system scales?
- How can we maximize availability for actively used data?
- How can we cope with the accelerated rate of data creation compared to storage

creation?

- How can we best allocate limited resources to different groups to manage cost?

There are certainly other issues in large scale systems, performance and security foremost, but solving these four problems will put us well on our way towards building systems with the capacity we need for future data.

Typically, research at this scale has been confined to talking about archival systems. Traditionally, systems have been classified as archival if they are designed to hold information for a potentially limitless amount of time and they have a "write-once, read-maybe" workload. Though this is not representative of many real systems [2], much of the work done with these systems in mind also applies to archival-by-accident systems, which have archival characteristics for most data most of the time but contain changing pockets of highly desirable active data. In this section, we present current work in managing large scale and archival systems with respect to the questions we wish to answer.

2.3.1 Power

Power is a prime scalability concern for storage systems. As systems add more storage, the costs of powering disks skyrockets and the corresponding cooling costs rise proportionately; though this may be changing [73], up until recently the disks tend to consume more power than the rest of the system combined [35, 10].

To control power expenditures in large scale systems, the most straightforward approach is to power down disks. The pioneering project in this area was MAID, which leaves disks idle when they were not in use [17]. MAID, however, works best when accesses are few and far between, so the power saved offsets the increased power consumption and loss in disk lifetime from having more spin-ups. e-RAID attempts to bypass this issue by creating intervals of idleness in RAID-1 systems by algorithmically balancing the load between the primary and mirror copy [118]. PDC attempts to concentrate popular data together to reduce disk spin-ups, but ultimately suffers the same fate

as MAID when presented with long tail accesses [80]. Our work addresses the main issue that all of these share, namely that on normal systems, data that is not in cache leads to scattered disk or track accesses. The additional disk activity negates many of the benefits of trying to keep disk activity low when not in use. By arranging data together according to use, regardless of how popular the data is, we can avoid enough of these additional disk accesses to see potentially large power savings (Chapter 5)

Greenan *et al.* explore power aware coding as another means of power management at scale [40]. Though I/O heavy workloads are known to be bad candidates for any technique that exploits idle time [14], Narayanan *et al.* show that significant idle periods exist in enterprise workloads, indicating that our technique may apply outside archival systems [75]. They go on to say that access stability could be exploited by grouping similar files together, but leave the area open as future work. Performance has been the primary focus for a number of power aware systems that have looked at access history and frequency [26, 106]. While performance is a concern for us, by calculating groups in the background we need to worry less about performance impact of the rest of the system design. Though Ganesh *et al.* were successful in using grouping to reduce power in an LFS-based system [35], LFS is not suitable for many real workloads [66]. We hope our approach will generalize across filesystems.

Though we are mainly interested in large scale systems, we build on a significant body of work that shows that data arrangement has a strong impact on the energy footprint of single-disk systems [29, 89, 33]. PARAID [119] similarly powers down multiple disks during periods of light system load. Much of this work can be translated to systems at the peta and exascales. Helmbold *et al.* point out that for mobile systems, an adaptive disk spin-down rate is superior to a fixed rate [43]. Zhu *et al.* take this one step farther to show that with careful data layout, multiple speed disks could significantly lower the power footprint of a system [84]. Laying out data on disk according to working sets should help a multiple speed disk system not split up working sets between different speeds of disk.

2.3.2 Failure Management

A driving application and motivation for our data grouping work is failure management, particularly through fault isolation. When a device fails in a large scale storage system, the impact of the failure is highly variable. Frequently, no one will even notice the failure until a scrubbing process or other external monitoring tool detects it. Latent failures are especially susceptible to this [64]. However, other failures of equally small size lead to organizational paralysis. For instance, losing one of a set of inputs to a process could make the entire process unusable until the data is restored, a process that is taking increasingly longer as the amount of data stored on these systems increases [60]. If data were arranged on disk according to semantically meaningful groups, a hardware failure would possibly affect many fewer projects, greatly increasing perceived system availability.

The concept of localizing data for fault isolation was first proposed by the D-GRAID project at the University of Wisconsin, Madison [100]. D-GRAID proposed to analyze the reliability increase of distributing blocks of a file across as many disks as possible versus keeping the blocks together on not only the same disk, but the same track. They proposed that since a head error is likely to affect adjacent track members more than random blocks on a disk, writing a file consecutively was an effective way to localize failures and thus minimize the functional cost of rebuild. The functional cost here is defined as the person or project time lost as a result of file unavailability during the rebuild process. Our grouping methodology will allow for failures to be localized to working sets, which represent use cases, allowing more of the system to be usable in case of failure.

Many large scale systems are provisioned to hold data for long, potentially archival periods of time. As systems scale, the concept of a "long" time to store data shortens. For instance, an exabyte scale storage system with 99.999% annual data retention will lose data in a decade comparable to what a petabyte scale storage system loses in a century. While the percent remains constant, the value of data does not necessarily decrease as more of it is stored. Therefore, the point at which advanced reliability measures must

be taken to ensure availability in a large storage system is likely to be sooner than we are accustomed to. Several studies have shown that storing data reliably over the archival time scale presents additional challenges, such as collected latent sector errors and data migrations, to the already difficult field of storage reliability [97, 64, 19, 91]. A challenge of this size requires combining known techniques of enhancing reliability with methods optimized for the expected workload and time scale of archival storage.

Recent work has shown failures remain an issue for large disk-based storage systems. For example, Schroeder and Gibson showed that the average disk replacement rates in the field are typically between 2 and 4%, implying a high reconstruction rate [96]. Other studies have shown that latent sector errors can lead to drive failure [7]. Pinheiro *et al.* showed that failures are both difficult to predict and common [81]. Scrubbing techniques can efficiently catch these otherwise unobserved errors so that the disks can be rebuilt promptly, but it is still often necessary to rebuild much of the disk [99].

Managing availability in RAID-like systems has received attention commensurate with increasing disk size and corresponding rebuild difficulty [55]. Disk sizes have grown, but the read speed is limited by power concerns and by the areal density of data on the platters [125]. Additionally, online-reconstruction is increasingly becoming CPU-bound, meaning that the cost of on-line reconstruction is unlikely to go down any time soon [63].

Modern storage systems can use parity to serve requests for data on a failed disk while the disk is re-building [44]. Many reconstruction optimizations have been proposed for distributed RAID rebuild [82, 83]. On-line reconstruction, serving accesses by reconstructing data on demand, has been shown to be between 3 and 70 times slower than serving requests from disk because of disk thrashing [126].

WorkOut addresses this degradation by employing a surrogate RAID array and using that to serve requests for "popular" data where popular is defined as accessed twice within the course of the rebuild [126]. Tiair *et al.* also base their reconstruction algorithm on popularity [112]. These approaches are limited to workloads that have many repeat accesses to the same data in a very short period of time. In PERSES, the grouping-

aware layout we discuss in Chapter 6, we sidestep this limitation by exploiting correlation along multiple dimensions in addition to recency and frequency. Other papers such as Thomasian*et al.* have shown significant improvement in performance during rebuild by modifying the underlying parity structure or scheduling [112, 48, 111]. PERSES could be combined with existing optimizations for reconstruction to evaluate the combined impact, but it is likely that since PERSES is primarily a data allocation scheme, it can be combined with all of these techniques to further reduce the impact of failure events.

High-performance storage systems such as Ceph and GPFS that add reliability through mirroring are designed for systems where availability and performance trump costs [94, 120]. For a long term system, keeping several times the number of disks you have for data on hand is infeasible from a cost perspective, but many still need reasonable performance. Coded systems such as RAID-5 [15] or even RAID-6 [24] reduce the disk overhead while distributing the risk for individual blocks of data, but do not provide enough reliability. Efforts have been made to use different erasure coded structures, but these still distribute failures more evenly than we believe is optimal for availability [41, 108, 124, 128]. Multiple copies of data can balance the loss of reliability that usage based data rearrangement will lead to. We believe that the additional cost will be manageable, but there is much work to be done in this area. GFS [90] shows that adding replicas is a viable tactic for adding in reliability. Disk-based systems such as Oceanstore [56] and SafeStore [85] combine mirroring, erasure codes, and strategic data placement to add reliability to their storage networks. While these systems are fast and highly available, they are not optimized to be low power and low cost. Other tiered reliability schemes similarly lack an emphasis on cost and power management [39, 74].

Finally, any grouping-based failure management scenario must work in the confines of the system's failure detection mechanism. While complete failures (*e.g.* a broken disk) are easy for the controller to spot, latent errors require active work to detect. Disk scrubbing, the canonical method of detecting latent errors, could potentially be even more effective with a grouping-driven priority queue [99, 108]. Biasing the algorithm for data layout is likely to increase network congestion on a point to point failure detec-

tion system such as Gupta *et al.* propose [42]. An alternative is accrual failure detection, which decouple monitoring and interpretation of failure events, allowing the load to be spread [92].

2.3.3 Data Reduction

Power and reliability both become more manageable if there is less data stored on the system. In a certain light, compression – the typical computer science answer to data reduction – is highly specialized data de-duplication with very small chunks [18]. De-duplication and compression have been the primary ways for system managers to losslessly reduce the amount of data on a storage system. However, de-duplication is plagued by the *data bottleneck problem*, which is the fact that de-duplication indices can easily grow too large for memory, forcing the index to page to disk. This is even more important in de-duplication for primary workloads where de-duplication is part of the I/O stack and thus directly impacts user perception of speed. We hope to address this problem by keeping two de-duplication indices, one on disk and one in memory, and pulling in fingerprints from disk to memory based on the groups of recently accessed elements.

Efforts that address the disk-bottleneck problem have typically been limited to backup systems, though Mandagere *et al.* demonstrate the type of trade-offs that are typically made to limit a deduplication index to available memory, mainly by increasing the chunk size [69]. Srinivasan *et al.* [104] propose to improve primary deduplication performance with iDedup by only duplicating chunks with high spatial locality. They also use temporal locality to restrict their in-memory cache of hashes to LRU. Essentially, they limit the blocks they de-duplicate to blocks that are hot and sequential. While this may be necessary for workloads with extremely high IOPS, we show in this work that it is possible to de-duplicate every block instead of restricting ourselves to blocks which have a high probability of duplication in certain workloads. Our technique is broadly applicable and results in better space savings than iDedup.

The Extreme Binning project tackles the data bottleneck problem by noting that

file similarity can be determined by comparing the IDs of a subset of chunks, allowing similar files to be grouped together in backup workloads by subsampling larger pieces of a stream [11]. Our approach aims for the same ends with different means. First, we assume we do not know about existing chunk to file relationships when grouping our data. Our groups are purely based on chunks. This is beneficial because it allows us to proceed with less system knowledge in an environment where many accesses are not sequential. Adding groups to the file similarity metric in Extreme Binning could improve their results on primary workloads. A similar technique used by Lillibridge *et al.* addresses the disk bottleneck problem by breaking up the backup streams into very large chunks and selectively de-duplicating them against similar chunks stored in a sparse index [62]. Though their throughput is very high, they rely on the large similar blocks of data that are common in backup workloads but generally absent in multi-user primary storage.

Primary de-duplication is an up and coming area, so there is little background work that discusses primary storage in a large scale system. Researchers such as Chambliss [18] and Storer [107] have began to use de-duplication on primary storage in addition to backups and archival systems. All de-duplication becomes difficult at the petabyte scale [32], but grouping has been used by some projects in backup systems. Efstathopoulos and Guo found that pulling in data according by group had a significant impact on the memory requirement of the index cache [32]. However, their method of group detection, relying on the spatial locality of the data stream in a backup workload, does not carry over to the random accesses of a primary de-duplication workload. Zhu *et al.* provide a comprehensive look at data de-duplication on backup workloads in addition to introducing the use of Bloom filters to improve the lookup speed for testing whether a write is a duplicate [132]. While this method is much faster than a linear search over the database, it is still not fast enough to avoid affecting performance. DEBAR improved the scalability and performance of deduplication for backup systems by aggregating a set of small I/Os into large sequential blocks after passing them through a preliminary filter [129]. Our work captures similar sequentiality through working set identification

and thus does not need to rely on the backup stream.

Srinivasan *et al.* [104] propose to improve primary deduplication performance with iDedup by only duplicating chunks with high spatial locality. They also use temporal locality to restrict their in-memory cache of hashes to LRU. Essentially, they limit the blocks they de-duplicate to blocks that are hot and sequential. While this may be necessary for workloads with extremely high IOPS, we show in this work that it is possible to de-duplicate every block instead of restricting ourselves to blocks which have a high probability of duplication in certain workloads. Our technique is broadly applicable and results in better space savings than iDedup.

2.4 Tiered Storage and Data Layout

Hierarchical or tiered storage can be considered as a grouping where the data is typically split either by domain specific characteristics or frequency of use into a small number of set partitions. Many of the techniques used to move data between tiers can be modified to decide how long to keep working sets and how to decide working set precedence.

Once we have groupings calculated, the next step is typically to lay the data out on disk based on the group or groups individual blocks belong to. This has analogs to tiered storage, where data is placed in different physical or logical tiers based on factors such as content and popularity. Tiered storage systems can be implemented using a natural split such as placing metadata in a separate device [108] or by placing more recently used data in a "better" tier of storage [39]. Tiered storage has been investigated on several different workload types including sensor networks, high performance computing, and archival systems for separation of different types of data [23, 120, 108].

All of these projects share a goal of laying out data optimally for a particular hardware configuration and workload. Typically, the focus is on bringing the most recently active data to the best tier of storage, such as PROFS which does this for LFS [117]. However, many of the same techniques could be used for re-arranging data efficiently

based on dynamic groupings. Many storage systems are also designed with some level of replication. Since blocks can belong to multiple working sets, there is a question of which working set or sets to write the block as a member of. In systems with replication, there are already multiple copies of blocks, allowing us to arrange several groups on disk per block. A data layout method along these lines used by Yahoo!® is PNUTS, which tries to locate replicas of data near the geographic location where they are most likely to be accessed [20]. Amur *et al.* look at layout for multi-replica systems in the context of power management [4]. Lamehamedi *et al.* look at the cost and reliability tradeoff of adding replicas, which we use to inform our later calculations about the benefit of storing multiple copies of files that are members of multiple groups [57]. Sun *et al.* show that for parallel systems, there is a strong argument for application-driven data placement [109]. With learned groups, we can exploit some of the same conditions without needing an out-of-band method of identifying which files belong to which application.

2.5 Caching and Prefetching

One way to frame our grouping techniques is to think of the storage system as a cache of arbitrary size. Grouping can provide many of the benefits of traditional memory caching without having to worry about expunging elements. Caching can be defined as looking for groupings of data that are likely to be accessed soon, based on any one of a number of criteria. Caching algorithms can even be adaptive and pick a cache criteria based on what provides that best hit rate [5]. The cache criteria can involve file or block grouping [80], but typically only in the context of grouping together popular or hot blocks of the system [117]. This is necessary because cache space is precious, so placing related, but cooler data into the cache would only serve to pollute it [133]. DULO biases the cache towards elements that have low spatial locality, increasing program throughput, but is affected by cache pollution issues for data that is rarely accessed [46]. Unified Buffer Management [51] can detect accesses with regular inter-reference intervals and pre-existing spatial locality in cache, but does not record relationships for future use.

We have relatively infinite space to store and reallocate groups, allowing us to store a record of past correlation in even rarely used data.

That said, results for caching are relevant for us since grouping works on a similar mechanism. Several projects incorporate caching into large scale storage. These caches can be based on temporal locality [17] or file or block-level popularity [80]. These studies show that for systems with low request rates, caching significantly improves the power efficiency of the system. Adding caches for both writes and reads to a grouped system should only improve the power efficiency [75]. One major vulnerability of our system is that a small group of files in disparate access groups might start getting accessed repeatedly for a burst. In this case, we could follow the lead of PDC [80] and use a multi-queue caching scheme to keep popular files or access groups in a persistent, always-on cache. Examining the tradeoffs between putting popular access groups in cache versus leaving them spinning on access could be especially interesting for workloads without regular access patterns, where we expect to not have many periods of low disk activity.

Kroeger and Long examined using a variant of frequent sequence matching to prefetch files into cache [54, 53]. Their work provides strong evidence that some workloads (SpriteFS traces, in this case) have consistent and exploitable relationships between file accesses. We are targeting a different problem, though with the same motivations. Instead of deciding what would be most advantageous to cache, we would like to discover what is most important to place together on disk so that when the cache comes looking for it, the data has high physical locality and can be transferred to cache with minimal disk activity. We assume that our methods will be used alongside a traditional cache because they complement each other, and it has been shown that both read and write caches amplify the benefits of grouping [75]. Diskseen performs prefetching at the level of the disk layout using a combination of history and sequence matching [26].

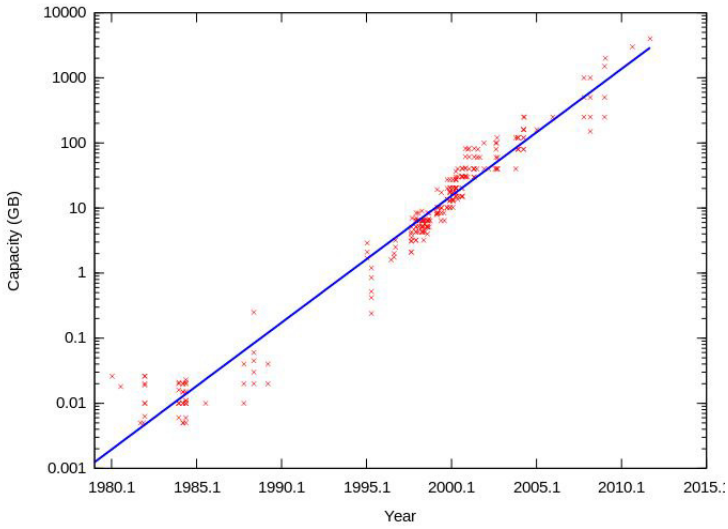


Figure 2.1: Disk capacity increases exponentially over time. Image Source: Wikipedia: Kryder's Law

2.6 Storage Media

Disk is the main storage medium for data because of a combination of low cost, history, ease of use, and high capacity. Though many other types of storage exist that is either lower cost, such as some tape devices, or higher reliability and lower power, such as flash, disks exist at a sweet spot that is likely to remain relevant for large scale, semi-actively used systems for some time. Both individual disks and storage systems are getting larger quickly. Figure 2.1 illustrates Kryder's Law showing that the areal density of hard drives has increased almost exponentially over the last decade. Many types of storage class memories are on the horizon, but none have as of yet come to market at sizes that could compete with disk and are unlikely to achieve comparable cost per gigabyte by 2020, when Kryder's Law projects that a 2.5" 14 TB hard drive will cost $40 [55]. Cheap, plentiful storage has already led to storage systems with large quantities of colder data that is arranged haphazardly and leads to a large seek overhead when accessed, leading to the archival-by-accident scenario we discuss in Section 2.2.1.

Our work was inspired by previous projects that focused on specific workloads or ap-

plications where grouping could provide a benefit. The DGRAID project demonstrated that placing blocks of a file adjacent to each other on a disk reduced the system impact on failure, since adjacent blocks tend to fail together – *e.g.* as a result of media imperfections or high-fly writes – and co-location of data thus limits the cross-project impact of failure [100, 95]. Schindler *et al.* show the potential gain from groupings by defining track-aligned extents and showing how careful groupings prevent accesses from crossing these borders [93]. They also demonstrate the prevalence of sequential full-file access, and make a strong case for predicting access patterns across data objects in environments with small files. Since most files in a typical mixed workload are still small [110], extensions to their work apply to wide array of real workloads. This thesis attempts to achieve similar results in a variety of domains using lighter weight grouping techniques.

With the advent of Solid State Drives (SSDs), the focus of localization efforts has shifted from metadata management to accesses: we seek to minimize the delay to the next accessed element and therefore reduce group fragmentation within the data users access to perform tasks. Though SSDs do not suffer the seek overhead of physical disks, reducing fragmentation can still provide benefits for power iiand reliability [59].

2.7 Discussion

There is a rich background of work in methodologies to detect working sets for specific workloads, and we build on their experiences gratefully. Our key differences from prior grouping work are a lack of reliance on domain knowledge or semantic metadata and an adaptable grouping system that can handle dynamic multi-application workloads.

The problems of disk based storage fall into the major categories power, reliability, and cost. Designing a storage system involves balancing these hardware-specific issues with a desired level of performance and availability. This balance has shifted as the size and purpose of storage systems has evolved. What began as a single monolithic storage system to serve an entire laboratory migrated to individual or per-project desktop storage

all the way back around to the cloud model, where a massive array of disks serves data to many varied users and projects.

Hierarchical storage models tackle this scalability issue by serving some requests out of a limited-size faster tier of storage. Current work in caching and prefetching help determine what to place in these tiers, but this work is limited by outdated models of storage system usage, such as hot-cold storage separation, or by a level of domain tailoring that is incompatible with the dynamic nature of multi-user multi-application workloads.

In the next several chapters, we detail how we do grouping followed by specific applications of our grouping techniques.

Chapter 3

Data Sets

Water, water, every where,
And all the boards did shrink;
Water, water, every where,
Nor any drop to drink.

– Samuel Taylor Coleridge

We tested our grouping methodologies and applications on a variety of datasets aimed at representing several classes of workloads. Much access data available to the research community is over five or even ten years old, and thus does not accurately represent how systems are being used in our cloud and always-connected world. To ensure that our results are applicable to real workloads, we focused on getting access traces that were collected recently – within the last decade – and that cover a variety of different workload types.

To analyze grouping, ideally one has the ability to gather complete block-level logs for a system with many users and many applications over a period of time commensurate to the dynamicity of a trace.Additionally, this trace would be collected before any file-system and hardware specific biases (*e.g.* write off-loading, sequential access removal) are introduced. Finally, having metadata or content ID to verify that elements that are selected to be in the same group have some semantic correlation is useful for group validation.

Finding traces with all of these attributes is difficult for researchers because of the privacy implications of rich metadata and the tracing overhead collecting large amounts of data on an active systems incurs. The datasets we use in this thesis are selected to provide as much breadth of workload type given what we had available. In this section, we provide basic statistics for all of the datasets we use in this thesis.

Table 3.1 outlines the I/O traces we used. Our goal was to model a variety of different use cases. In the following sections we provide details about each data set along with two-dimensional histograms showing the distribution of complete traces in space and time. Darker sections of the histograms correspond to higher concentrations of accesses.

3.1 MSR

Our first dataset, **msr**, represents one week of block I/O traces from multi-purpose enterprise servers used by researchers at Microsoft Research (MSR), Cambridge [75]. We chose these traces for two reasons: first, they allow us to simulate the bare bones block-timestamp trace we can collect from a protocol analyzer. Secondly, these traces were collected in 2007, making them more recent than most other publicly available block I/O traces.

3.1.1 Data Format

As of 2013, the **msr** traces are available from SNIA(`http://iotta.snia.org/`). We assume that block offset and access size form a unique, permanent identifier for a particular snippet of data. We realize this assumption does not hold true over the long term or in frequently re-written data. We classify blocks based on the difference in timestamps, block offset, and by the order they appear in the trace.

The offsets accessed in our data, which is from a single disk, were spaced between 581632 and 18136895488.

Table 3.1: Comparative Overview of Datasets

Name	Size	Type	Year	R/W	# Accesses	% Unique Accesses	Avg. IOPS	Max. IOPS	Length
msr	75MB	Block	2007	9/91	433655	3.26%	6.53	3925	7 days
ent-storage	2.8GB	Block	2010	*	2161328	44.82%	75997	342142	3 days
fiu	5.9GB	Block	2010	4/96	17836701	9.44%	40.49	20104	21 days
water	8.6MB	File	2007-9	100/0	92251	5.10%	0.15	16	792 days
wash	436MB	File	2007-10	100/0	5346868	39.70%	0.085	114	945 days

Table 3.2: Sample Data from MSR Cambridge Research Machines

Timestamp	Type	Block Offset	Size	Response Time
128166372003061629	Read	7014609920	24576	41286
128166372016382155	Write	1317441536	8192	1963
128166372026382245	Write	2436440064	4096	1835

3.1.2 Access Distribution

Figure 3.1(a) shows that the offsets are approximately uniformly distributed, which allows us to use a consistent distance metric across our search space.

This dataset was very write-heavy with a read/write ratio of 10:90. This ratio is almost entirely attributable to a small range of offsets that are likely to represent an on-disk cache, which is an anecdotally known feature of NTFS [71]. Figure 3.1 shows the accesses by block over time, and Figure 3.1(b) highlights the read activity. Removing the writes (Figure 3.1(b)) shows some dense areas possibly corresponding to data groups. Despite the cache spike, the accesses in our data are approximately uniformly distributed across offsets.

3.2 FIU

Our second trace, **fiu**, is from Florida International University (FIU) and traces researchers' local storage [52]. This is a multi-user, multi-application trace, with activities including developing, testing, experiments, technical writing, and plotting. The traces were collected in 2010 from systems running Linux with the ext3 file system. The **fiu** trace is our most diverse trace in terms of known applications. It also had the additional benefits of having content hashes for deduplication analysis and process information that we used to validate groupings.

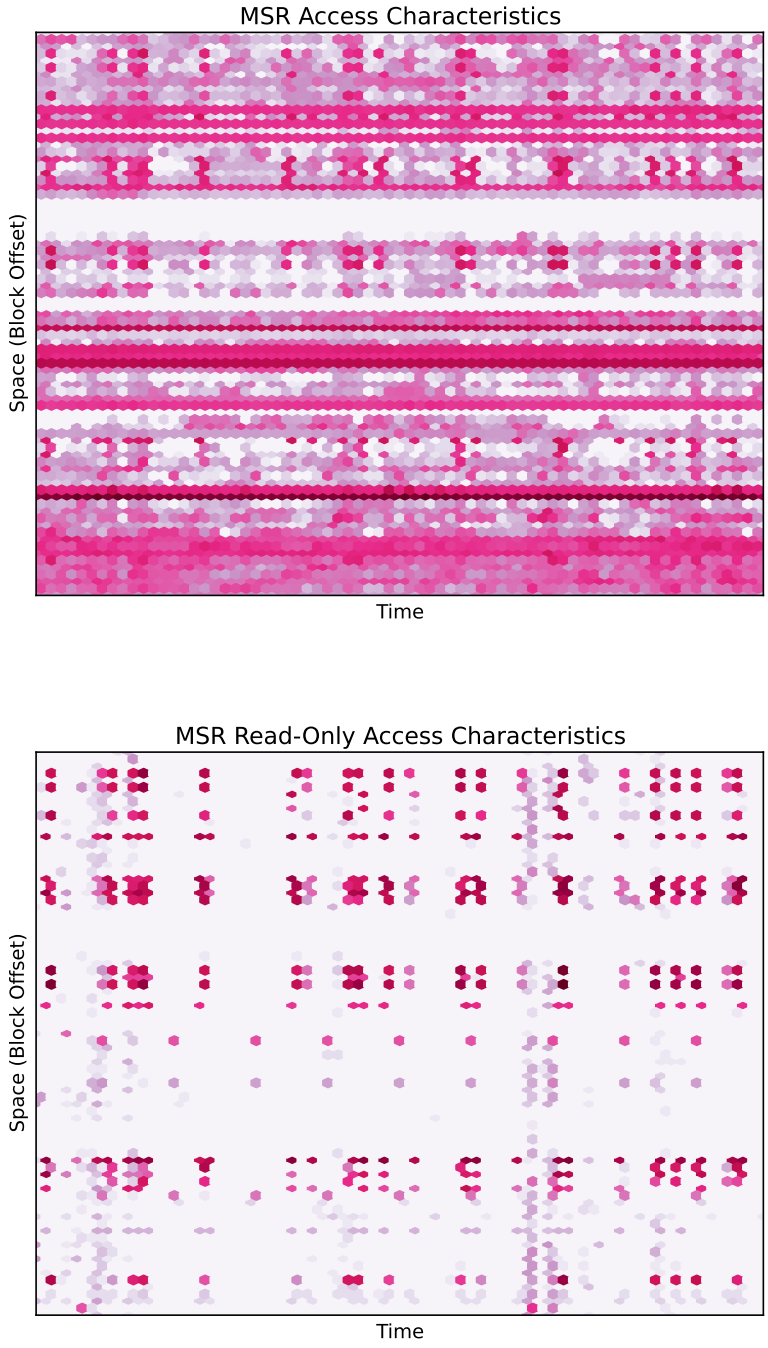


Figure 3.1: These 2-D histograms show the spatiotemporal layout of block accesses across the **msr** trace with and without writes. Darker bins correspond to higher access density.

3.2.1 Data Format

Table 3.3: Sample Data from Florida International University Research Machines

Timestamp	PID	Process	LBA	Size	R/W	Maj, device #	Min. device #	MD5
0	4892	syslogd	904265560	8	W	6	0	531e779...
39064	2559	kjournald	926858672	8	W	6	0	4fd0c43...
467651	2522	kjournald	644661632	8	W	6	0	98b9cb7...

To obtain a large corpus, we merge a collection of daily traces from FIU. Since the traces contain process information, we can use the data to externally verify the classifications we make by showing that data within a group was last accessed by the same process. Size is in units of 512 byte blocks, and the MD5 is calculated per 4096 bytes. We use LBA as the spatial component of the calculation; we see in Chapter 7 that LBAs shift over time, and discuss what that means for their groupability. As of 2013, this data is publicly available at `http://sylab.cs.fiu.edu/doku.php?id=projects:iodedup:start`.

3.2.2 Access Distribution

This dataset was also surprisingly write heavy, likely due to small system writes replicated per user. Over 33% of accesses were to duplicate blocks, determined by the MD5 hashes. Figure 3.2 shows that the writes are evenly spaced throughout the area where accesses occur; when writes are removed, the overall access layout remains similar. This also implies that writes are spread across multiple users.

3.3 Enterprise Storage

Our other block-based trace, **ent-storage**, is from researcher home directories at IBM T.J. Watson Laboratories. The directories are housed on an IBM XIV, an 80TB self-contained storage system that provides many features including mirroring, read look-

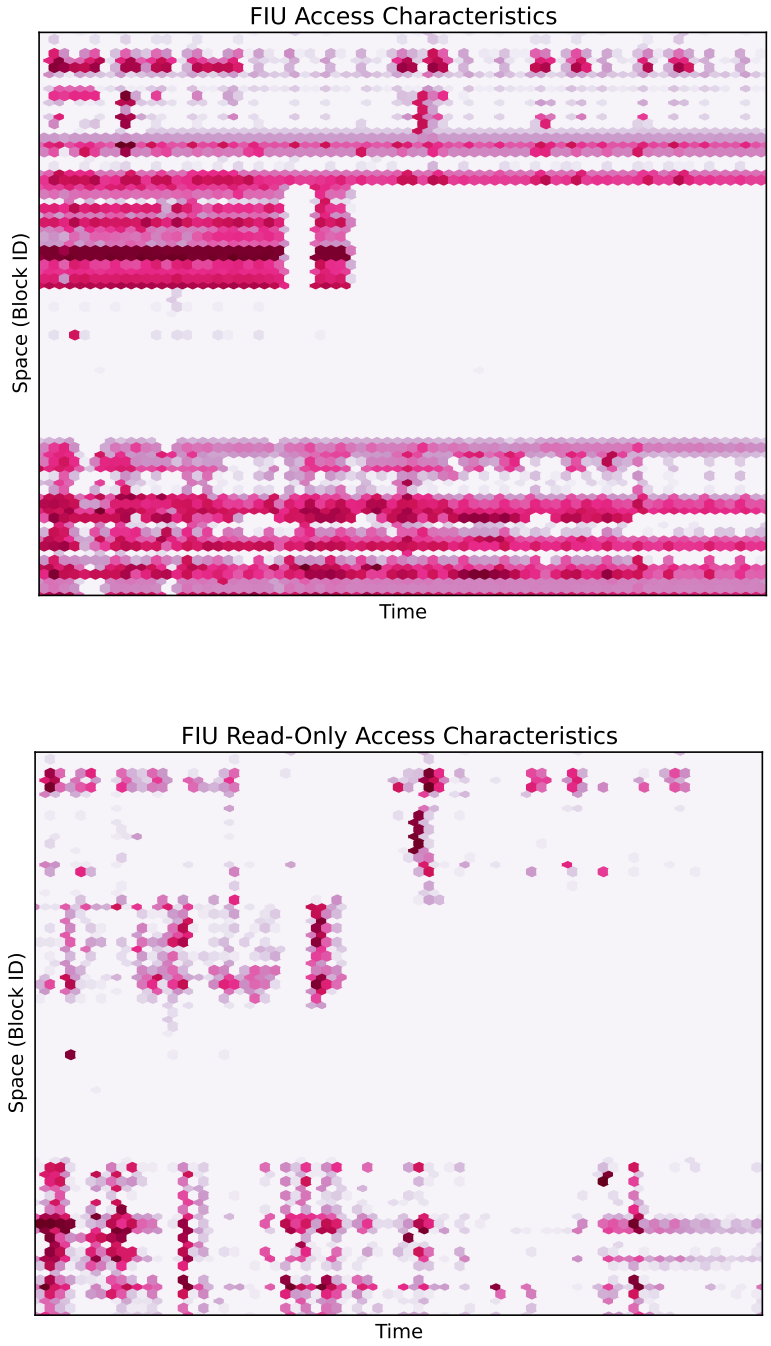


Figure 3.2: These 2-D histograms show spatiotemporal accesses for the **fiu** dataset with and without writes. Darker bins correspond to higher access density.

ahead, and 7TB of SSD cache [34]. The directories are stored under GPFS [94] and are used by over 100 researchers.

3.3.1 Data Format

Table 3.4: Sample Data from IBM Watson, stored on an XIV

kind	# blocks	is_read	LBA	time	volume	initiator_id	fingerprint
0	47	1	825850448	1313956791731167	101921	1000012	6c5fb8d...
0	61	1	825848704	1313956791765460	101921	1000002	d10b05c...
0	8	1	1485868928	1313956791817914	102669	1000009	76ca22b...

The XIV is split into multiple volumes, but LBAs are consistent across volumes and so still usable as a spatial reference. An additional issue with this trace is that the trace is post-adaptive lookahead. This means that sequential accesses are effectively removed from the trace by the system itself. This dataset is not publicly available.

3.3.2 Access Distribution

Figure 3.3 shows the access layout for **ent-storage**. We see that half of the LBA space is very sparse while the other half is relatively evenly covered. Since we have no metadata for this trace, we do not know why this occurs, but it could be related to the logical configuration of the specialized hardware this trace was gathered from.

3.4 Water

The first of our two file-archive access traces is from the California Department of Water Resources [2]. Our data consists of 90,000 accesses to a record store from 2007 through 2009. We make the assumption that queries correspond to record accesses, since each web query is likely to cause a disk access to the web server. The data set is pre-grouped, and the grouping labels we consider for each access are “Timestamp,”

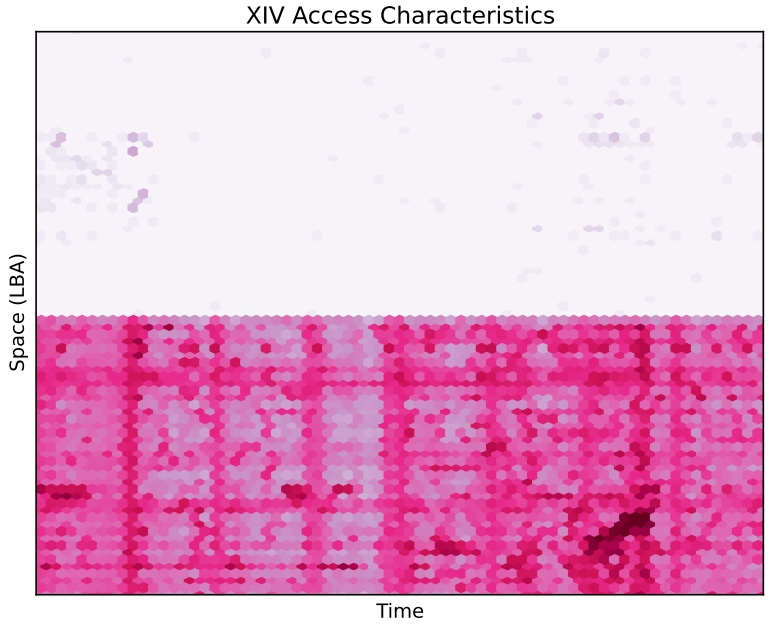


Figure 3.3: This 2-D histogram shows the spatiotemporal layout of block accesses across the **ent-storage** trace. Darker bins correspond to higher access density.

"Site," "Site Type," and "District." The dataset provided an additional grouping, "Year," that we chose not to use because it is inconsistently applied.

3.4.1 Data Format

Table 3.5: Sample Data from California Dept. of Water and Power

ID	Date	IP	Site	Type	District	File
1250	10/1/2007 13:49:15	72.166.5.82	16N2W5B1M_EXT	GW	ND	GROUND..._PLOT.PNG
1251	10/1/2007 13:49:16	72.166.5.82	16N2W5B1M_EXT	GW	ND	GROUND..._DATA.ZIP
1252	10/1/2007 15:08:36	72.166.5.82	16N2W5B1M_EXT	GW	ND	GROUND..._DATA.ZIP

The ID is unique per "visit", which corresponds to a user session, and "Type" is the type of site that the accessed file was collected from. The dataset does include complete filenames, but we truncate them here for space. Because the dataset is file-based and lacks directory metadata, we cannot form a spatial distance relationship between accesses. Instead, we rely on the pre-labeled metadata to explore what grouped data can accomplish on curated systems. This data is not publicly available.

3.4.2 Access Distribution

Figure 3.4 shows a 1-dimensional histogram of accesses per day across the course of the **water** trace. One interesting note is that the archive was dominated by search engine activity, demonstrated by the abrupt increase in access activity at about 700 days in the complete graph. The lower graph, which does not include the top six accessing IPs, is noticeably more uniform. We do not have a spatial component in this trace, so we cannot show a spatiotemporal histogram.

3.5 Washington State Archives

Our other file-archive data set is a database of vital records from the Washington state digital archives where records are labeled with one of many type identifiers (*e.g.*

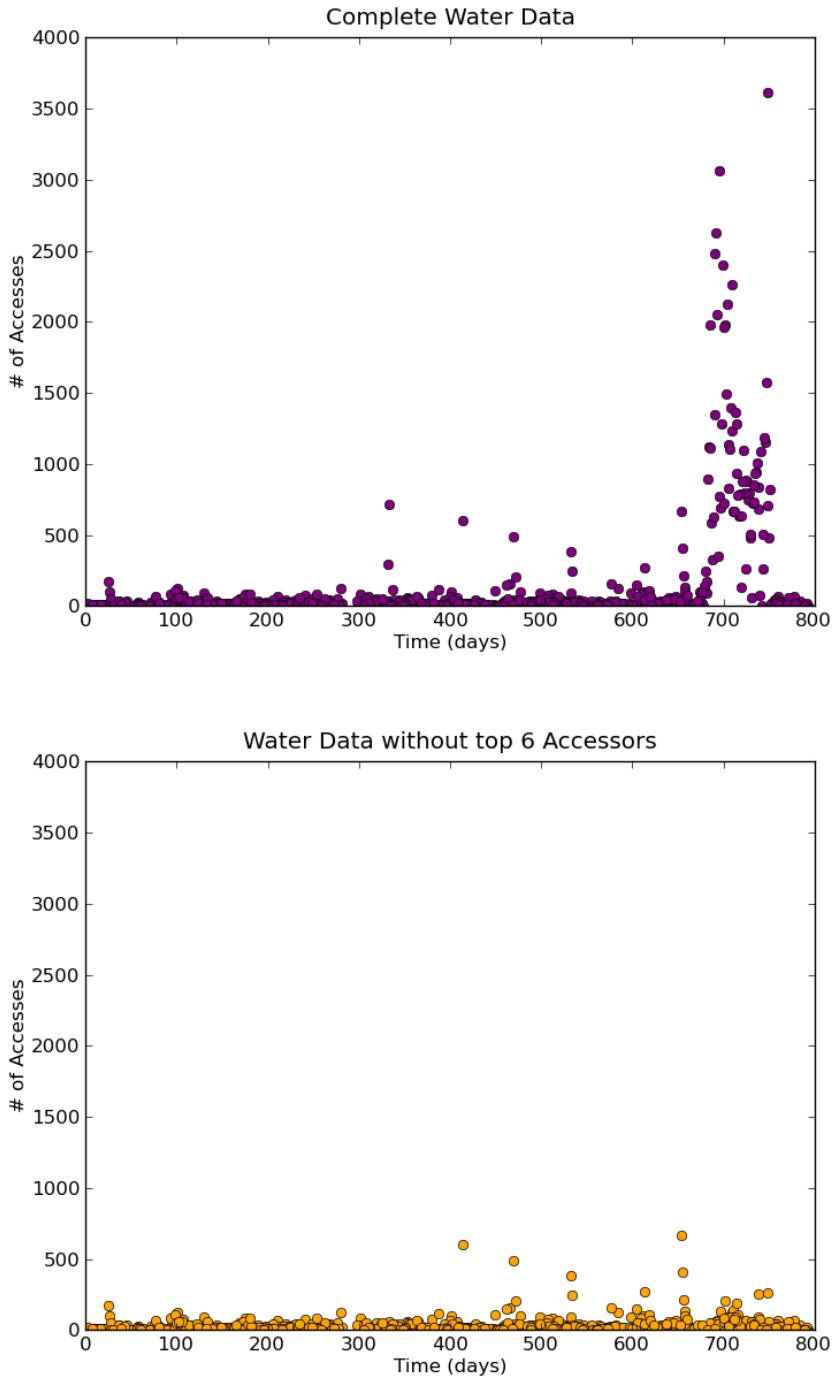


Figure 3.4: Access patterns for **water** with and without the top 6 accessors, based on IP

"Birth Records", "Marriage Records") [2]. We examined 5,321,692 accesses from 2007 through 2010 that were made to a 16.5 TB database. In addition to the supplied type identifiers, each record accessed had a static[1] RecordID that is assigned as Records are added to the system. We use these IDs as a second dimension when calculating statistical groupings.

3.5.1 Data Format

Table 3.6: Sample Data from Washington Department of Records

RetrieveTrackingID	UserSessionID	RecordID	RetrieveDate
1	{C8E99715-4725-427A-BCDF-708109D4935F}	34358	2007-09-27 13:31:10.407
2	{D2B7A983-7CC6-46C8-A10F-7B2557CF204F}	94267	2007-09-27 15:36:13.287
3	{1CE276B9-06F4-4AF7-9A08-E4038D83BBFB}	46679	2007-09-27 15:59:42.737

In addition to the access trace in Table 3.6, we also had a file that mapped most of the RecordID values to assorted RecordType values such as "BirthRecord," "MarraigeRecord," etc. We treat RecordType as a pre-labeled group for categorical grouping, but also use RecordID as a spatial dimension to statistically group the **wash** dataset. Though RecordID does not directly map to an on-disk location, we assume it correlates to ingress and assume that records are originally laid out sequentially by RecordID.

3.5.2 Access Distribution

In Figures 3.5 and 3.6, we show both the access frequency over time and the distribution of accesses in time and space, as defined by the RecordID field. There are clear gaps in RecordID space, which may correspond to either stale or missing records. Additionally, there is a large block of RecordIDs in Figure 3.6 that is accessed continuously for the middle of the trace, corresponding to the large number of accesses between 600 and 800 days we see in Figure 3.5. Unlike the **water** dataset, we see that removing the

[1] This is not quite true, but accurate for our purposes. Further explanation can be found in [2].

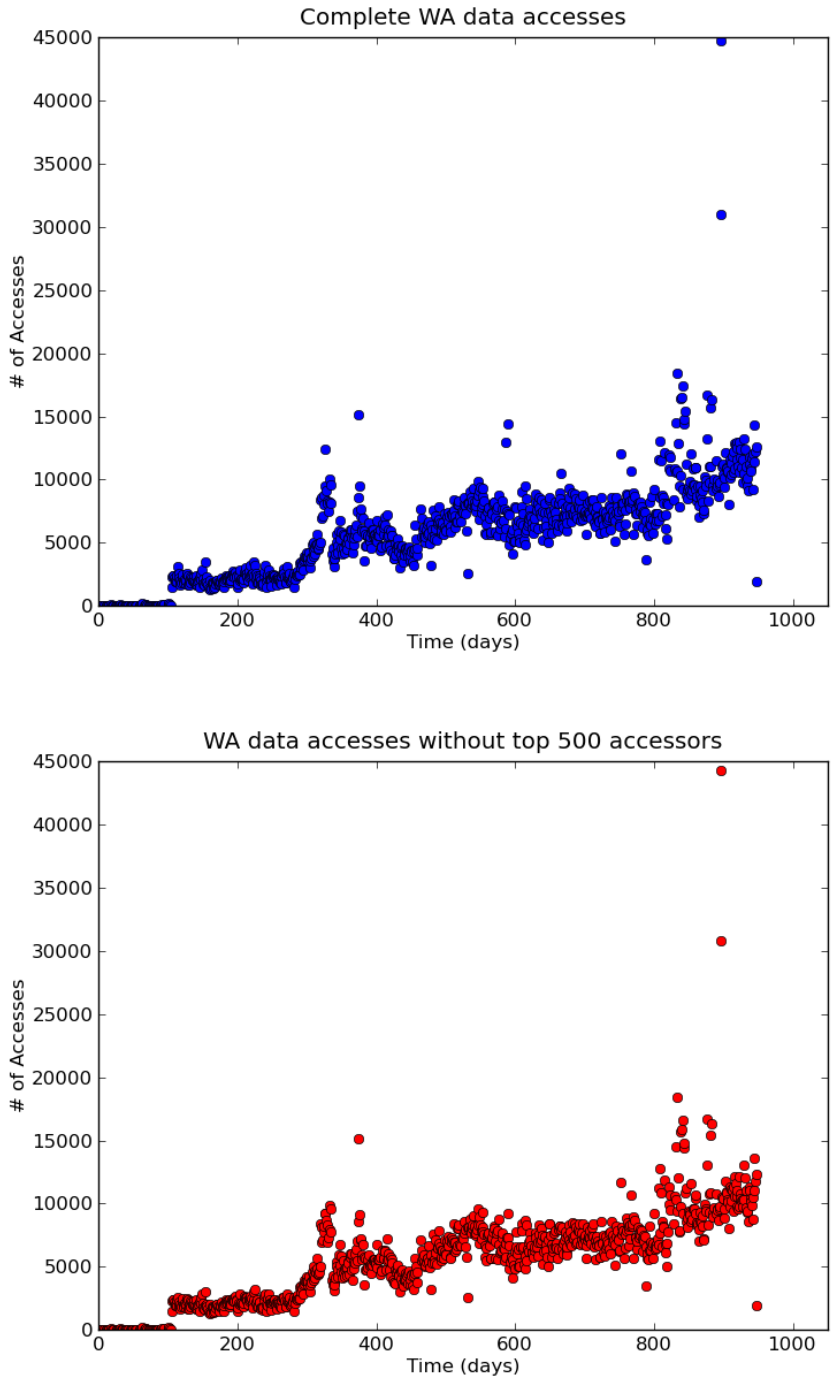


Figure 3.5: Access histograms for **wash** with and without the top 500 accessors by User Session

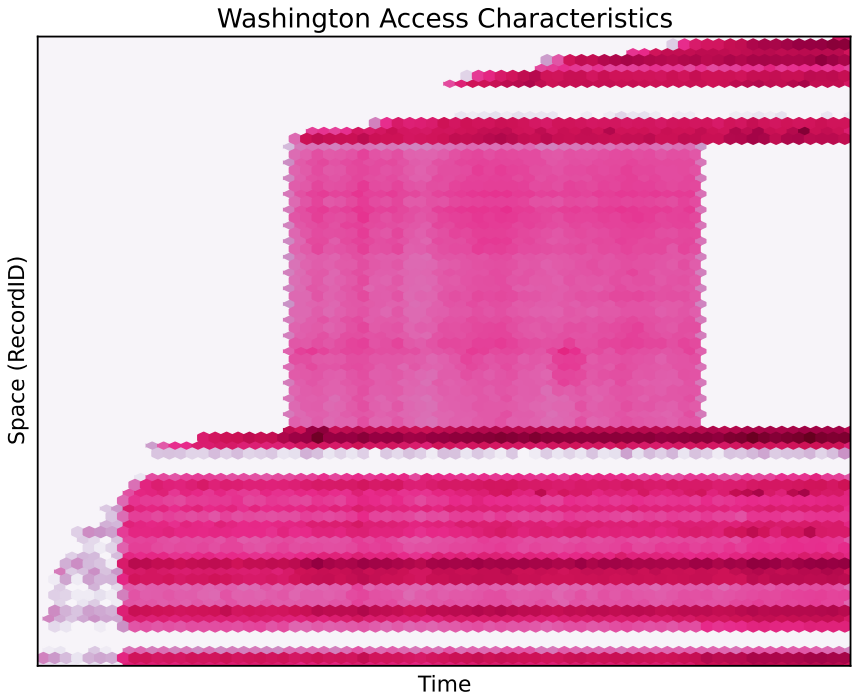


Figure 3.6: This 2-D histogram shows the spatiotemporal layout of block accesses across the **wash** trace. Darker bins correspond to higher access density.

top accessors has little effect on the access distribution over time. This is partly because user sessions are not as broad a cut-off as IPs, but we also have no reason to believe that this archive was externally searched. Further information about this data can be found in [2].

Chapter 4

Data Grouping

We're kind of finessing this through the back door...

– H.P.D. Michael Moody

Before we can arrange data according to probability of access, we need to understand how to identify groups in data. Grouping data is the necessary first step before we can leverage groupings for power savings, fault isolation, or any of a variety of other benefits provided by data grouping such as avoiding track boundaries [93], implementing SLAs, or doing intelligent data distribution in heterogeneous storage systems. The end goal of any grouping is to be able to predict future data usage, whether accesses or dependencies.

In an ideal world, we would be able to combine accesses based on either data contents of the application, user, or use case that initiated the access. We term this sort of information-driven labeling *categorical grouping*. Categorical grouping is a well studied problem, and thus for datasets with rich metadata we can treat the grouping problem as mostly solved and focus on the interesting applications [6]. If the level of domain information needed for categorical grouping can not be realistically obtained, we resort to *statistical grouping*, which is a collection of statistical and machine learning techniques that we have developed to identify groups with limited domain knowledge.

Unlike many techniques in the world of caching, we are looking for what elements a given element will be accessed with regardless of whether that element or any of its

fellow group members have a high chance of being accessed at all. We are relatively unlimited in the size of what we can, essentially, "pre-pre-fetch." While a group could be pulled into cache, a group could also be part of the same batched read request. By grouping data regardless of how often it is accessed, we hope to capture associations caused by the long tail of rare accesses that still occur in observable clusters. Once found, groups can be laid out contiguously on the storage media to increase efficiency. In our model, grouping is neither one-to-one nor onto; elements can be members of zero or multiple groups. Extant systems such as the Google File System [90] use distributed replicas to improve reliability, and many more systems could benefit from selectively replicating important data – given the idea that data that is part of multiple groups is more system critical than more special purpose data.

Previous work has argued that categorical grouping is the best option for domains with rich tracing and significant domain information [6]. Rich tracing is, however, difficult to come by. One silver lining of the massive amount of data modern systems create is that it is easier to train statistical learning systems on systems with high IOPS, since we have more data to support or contradict any prior calculations. Thus, we revisit the problem of statistical grouping and have found that on several workloads statistical grouping tracks real world usage patterns without adding excessive computational overhead. In addition to discussing statistical and categorical grouping techniques in further detail, this chapter outlines the approaches to group validity and scalability that we have taken and what real world scenarios our groupings may correlate to.

4.1 Statistical Grouping

Modern storage systems are moving back from a highly distributed, low IOPS environment to large shared storage servers that are under constant load. These systems are growing with the popularity of the cloud, as storage management is consolidated and heterogeneous data such as stables of virtual machine images, for instance, are stored together on a storage system accessed by disparate nodes [18]. In some systems, such as

systems designed for high performance computing, collecting rich metadata from storage accesses introduces an unacceptable amount of overhead in the form of additional disk operations. In others, groups need to be calculated in real time to provide benefits such as cache pre-fetching, so any grouping must be fast to obtain and process – ideally low dimensional. Both of these purposes are served by grouping using raw I/O traces at the block, file, or object level. On a real system, it is frequently impractical for security or performance reasons to put in hooks to collect even file-level access data. The classification into groups is then just based on spatiotemporal distance, as defined by the particular environment.

There are additional benefits to not using rich metadata or external domain knowledge when selecting groups. Both metadata and domain knowledge require upkeep by a local expert in the system workload. Over time, the logging methodology can change, leading to inconsistent interpretations for metadata fields [2]. Also, if an administrator leaves, their terminology and understanding of the system must be accurately transferred to their successor. Finally, the most important flaw with categorical, rich metadata systems is that usage patterns, especially in multi-user, multi-purpose storage systems, are constantly shifting. This makes it almost impossible to derive groups that have long term predictive capability based on categorical data.

Our statistical techniques are designed to create groups quickly to adapt to changing conditions while requiring a minimal amount of overhead. The tradeoff is that we will not be able to reach the same level of predictive accuracy that a domain specified grouping can get in its best case, where for example it is known exactly what the working sets will be. However, we have found that in most cases, the adaptability of statistical grouping provides better long term predictive capability to the groupings compared to static domain-based groupings, which mirrors earlier results that show that dynamic grouping has a lower overhead and higher value [16].

The statistical grouping techniques we have researched all use data that can be collected non-intrusively from a running system with minimal modification. For example, much of our work uses block I/O traces, which can be obtained by attaching a protocol

analyzer to the disk bus to watch the low level communication to the disks and reconstruct a block I/O trace from these patterns. In addition to alleviating privacy concerns, this type of data is straightforward to collect without impacting the performance of high performance systems. This technique has been successfully used to collect block I/O traces at Seagate [88].

Another reason we found that this was a better way to do trace-based prediction is privacy. We have found that obtaining data to do predictive analysis is easier if one can make a clear argument that sharing data will not create any privacy concerns for the source organization. After well documented cases of failed or insufficient data anonymization, such as the infamous AOL data leak [9], companies are very aggressively defending internal privacy to the detriment of well meaning researchers. This concern is part of the reason much modern research in predictive grouping uses data five to ten years out of date if they use real data at all [1].

4.1.1 Calculation

Our statistical classification scheme has two components: the distance metric used for determining distance between data points and the partitioning algorithm that identifies working sets based on these distances. We offer three different partitioning algorithms and explain how each could fit a particular type of workload and environment.

Most of our statistical data is two-dimensional, consisting of a spatial location and last access time. We had access size data for some traces, but found that groups were more consistent if accesses of various sizes are broken into homogeneous chunks. For block I/O traces, we treat the block offset as a unique identifier for a location on the physical disk. We found that even though this offset can refer to different data over time (Fig: 4.1), there is enough information in the offset to classify with. This could in part be because the usage of data can remain similar even when parts of a file are overwritten. The uniqueness assumption for block offsets generally holds for application files and other static filesystem components, though it will break down for volatile areas such as caches or log structured file systems. As we see in Figure 4.1, the spatial dimension

can map well to the data for some time. Making this assumption allows us to use very sparse data for our analysis, since spatio-temporal data is ubiquitous in dynamic traces and, when the spatial component can be treated as a unique ID, it can be used to classify data.

We now cover our two grouping components, the notion of "similarity" and the partitioning algorithms, in more detail.

4.1.2 Similarity

Defining a distance metric that captures the relationship between disk accesses without over-fitting or underestimating relationships is crucial to forming meaningful groups. Focusing only on the trace allows us to adapt as the use of the system evolves. We offset this adaptability by using spatial distance, which has the benefit of keeping our groupings resistant to noisy changes: two offsets grouped together are biased towards staying grouped together. Finally, this method of characterization could expose previously undetected high level activities such as undeclared application dependencies or sudden behavior changes implying a security event.

Before we can form groups of similar accesses, we need to define what it means for two disk accesses to be similar. We can statistically group any data with at least two dimensions that have a well defined distance metric. In this work, we have used LBA-timestamp pairs, record IDs with access times, and block I/O traces. Throughout this section, we discuss similarity in the context of block I/O traces, but the same techniques apply to our other statistically grouped data.

Block I/O traces have two dimensions: *time*, the timestamp of the access request, and *space*, the location of the data being accessed. To compare accesses in these dimensions, we need to define an effective distance metric over time and space that has few parameters and is fast to compute. Though there are many possible choices, we have focused on variants of Euclidean distance so far for simplicity and generalizability. For richer datasets, we can also include features such as user requesting, user created, process requesting, size, etc. With more dimensions, we have a wide array of statistical

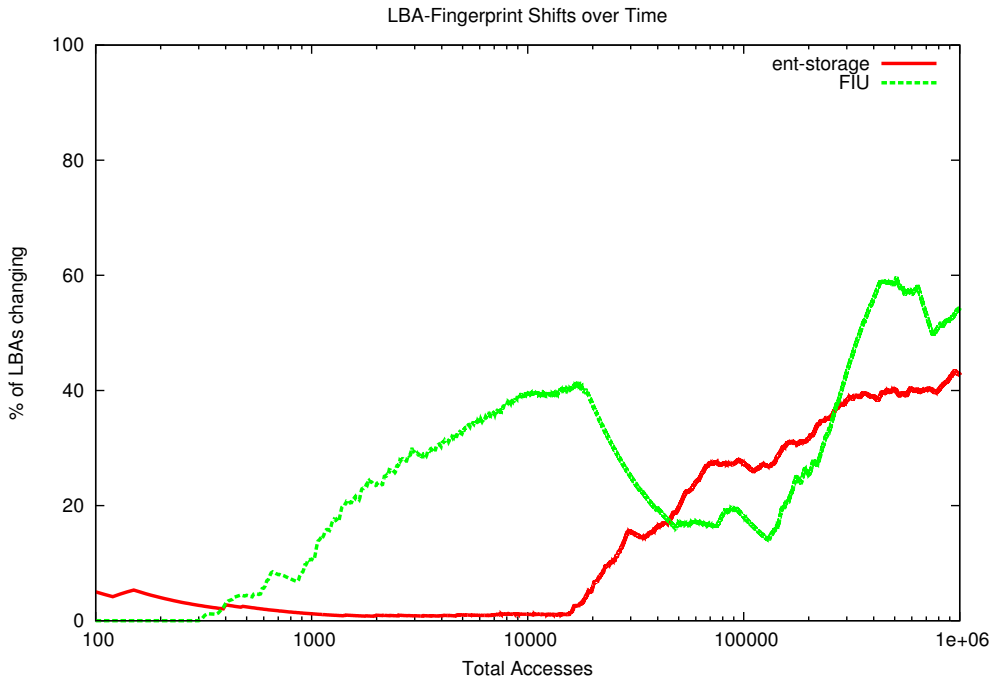


Figure 4.1: An example of offset-data mismatches over time. Logical block addresses are used in place of physical offsets, and each LBA corresponds to a data fingerprint. In the enterprise dataset, `ent-storage`, labels remain consistent for many accesses before the % changed creeps up, while in the research university dataset represented by `fiu` the mapping was more volatile. We re-calculated groupings after about 250,000 accesses because predictivity was dropping, partly due to the mismatch. Surprisingly, the `fiu` workload ended up with more predictive groups than the `ent-storage` workload. We discuss this more in Chapter 7

similarity metrics available such as the Sørensen similarity index [103], which biases against outliers, or Tanimoto Distance [45], which provides a set comparison that is optimized for grouping seemingly dissimilar sets [67].

4.1.3 Calculating Block I/O Distance

Our partitioning algorithms depend on a pre-calculated list of distances between every pair of points, where points each represent single accesses and are of the form $\langle time, offset \rangle$. We experimented with using points of the form $\langle time, (offset, size) \rangle$,

but we found this decreased the signal to noise ratio of our data considerably. In a dataset with more fixed size accesses, using $(offset, size)$ should result in a tighter classification.

In production, our system is designed to look at trace data in real time. This introduces an inherent bias towards accesses that are close in time versus accesses close in space, since accesses close in time are continuously coming in while accesses close in space are distributed across the scope of the trace. Intuitively, this is acceptable because the question we are trying to answer is "are these blocks related in how they are accessed," which implies that we care more about ten points scattered throughout the system that are accessed, repeatedly, within a second of each other than we do about ten points that are adjacent on disk but accessed at random times over the course of our trace.

4.1.3.1 Distance Matrices

For most of our grouping methods, we use a pre-calculated list of distances between every pair of points.

We create two different types of distance lists. The first is a simple $n \times n$ matrix that represents the distance between every pair of accesses (p_i, p_j), with $d(p_i, p_i) = 0$. We calculate the distances in this matrix using simple weighted Euclidean distance, defined as $d(p_i, p_j) = d(p_j, p_i) = \sqrt{(t_i - t_j)^2 + oscale * (o_i - o_j)^2}$ where a point $p_i = (t_i, o_i)$ and the variables are $t =$time, $o =$ block offset, and $oscale$ is an IOPS-dependent weighting factor on the block offset. As IOPS increases, the information in the current location of blocks decreases (Figure 4.1 shows the loss of information over time for **ent-storage**, a high IOPS workload, vs. **fiu**, a low IOPS workload), so a lower value of $oscale$ should be used.

Figure 4.4 shows average group sizes across the entire parameter space for an NNP grouping, as described in Section 4.1.4.2. It shows that for a given workload, only a small range of $oscale$ values produce a non-trivial grouping. For the datasets we tested, only large variations of $oscale$ produced appreciable changes in the resultant groupings, implying that $oscale$ is relatively stable.

In this global comparison of accesses, we were most interested in recurring block offset pairs that were accessed in short succession. As a result, we also calculated an $m \times m$ matrix, where m is the number of unique block offsets in our data set. This matrix was calculated by identifying all the differences in timestamps $T = [T_1 = t_{i1} - t_{j1}, T_2 = t_{i1} - t_{j2}, T_3 = t_{i2} - t_{j1}, \ldots]$ between the two offsets o_i and o_j. After some experimentation, we decided to treat the unweighted average of these timestamp distances as the time element in our distance calculation. Thus, the distance between two offsets is:

$$d(o_i, o_j) = \sqrt{\left(\frac{\sum_{i=1}^{|T|} T_i}{|T|}\right)^2 + oscale * (o_i - o_j)^2}$$

4.1.3.2 Ranged and Leveled Distance Lists

Calculating the full matrix of distances is computationally prohibitive with very large traces and impossible in an online system. We need to handle real-time data where relationships within the data are likely to have to have a set lifetime, so we also looked into creating lists of distances between the most relevant pairs of offsets. To do this, we again bias towards offsets that are close in time. For very dense workloads, we suggest choosing a range r in time around each point and calculating the distances from that point to all of the accesses that fall in range, averaging the timestamps for accesses that occur with the same offset, as in the previous section. For real-time traces, the range has to be large enough to capture repeated accesses to each central point to reduce noise. Section 4.1.4.2 discusses one scalability approach we successfully used to handle traces with over 300,000 IOPS.

For static trace analysis, where groups do not need to be calculated quickly, we have the ability to paint a more complete picture of how a given offset is related to other offsets. Instead of calculating ranges around each point, we calculate ranges around each instance of a given offset o_i. We do this by calculating the distance list around each of N instances of the offset,
$rDist(o_{i1}) = [(o_j, d(o_{i1}, o_j)), (o_k, d(o_{i1}, o_k), \ldots]$. We then take the list that each instance returns and combine them. This gives us a better understanding of trends in our trace and strength of association. If an offset o_i appears next to o_j multiple times, we have

more reason to believe they are related. To combine the list, we first create a new list of the offsets that only appear in one of our lists – these being elements that do not need to be combined. For the remaining elements, we take the sum inversely weighted by the time between their occurrences. For example, say we have an offset o that is accessed twice in our trace, at times t_1 and t_2, with distance lists: $[(o, o_i, d(o, o_i)_1), (o, o_j, d(o, o_j)_1]$ and $[(o, o_i, d(o, o_i)_2), (o, o_m, d(o, o_m)_2]$. The combined distance list would then be:

$$[(o, o_i, d(o, o_i)_1 + \frac{d(o, o_i)_2}{|t_1 - t_2|}, (o, o_j, d(o, o_j)_1, (o, o_m, d(o, o_m)_2)]$$

This heavily weights offset pairs that occur near to each other, which results in dynamic groupings as these relationships change. Switching the inversely weighted sum to an inversely weighted average smoothes this effect, but results in groups that are less consistent across groupings.

If accesses are sparse, we set the range in terms of *levels* instead of temporal distance. A level is defined as the closest two points preceding and succeeding a given access in time. A k-level distance list around a point p_i is then the distance list comparing p_i to the k accesses that occurred beforehand and the k accesses that occurred afterwards. In sparse, static, traces, we use these levels to manage the tradeoff between computational power and accuracy. Therefore, our work sets a minimum k as the median group size and increases this value based on computational availability. The distance lists are calculated the same way as they are for a set range.

4.1.4 Partitioning Algorithms

The goal of all of the group partitioning algorithms we work with is to identify groups that have a high probability of co-access within a small amount of time. These groups could correspond to individual working sets in the data, but are equally likely to arise from system-wide trends. We are particularly interested in untangling groups that are interleaved in the disk access stream. Large, long term storage systems that grow organically also develop heavily interleaved access patterns as more use cases are added to the system.

Our distance calculations return an answer for the question "how similar is element a to element b?" With this similarity information pre-computed, we now look at the actual grouping of accesses into working sets.

4.1.4.1 Neighborhood Partitioning / N-Neighborhood Partitioning

Neighborhood partitioning is an on-line, agglomerative technique for picking groups in multi-dimensional data with a defined distance metric. It is the best technique for data with more than two dimensions: distances are calculated over all dimensions and then the grouping itself runs linearly in the number of points. We do not use ranged or leveled distance lists with this method to limit computational overhead.

The first step is to select a window size, w. The window size is the amount of data that is used to create a single grouping. In high IOPS workloads, we use several of these windows to classify based on local requirements. For example, in the HANDS implementation we discuss in Chapter 7, we selected $w = 250$ MB because of local memory constraints. NNP requires up to w^2 of memory to store a pairwise distance matrix between elements in the window. However, this matrix is typically sparse since, below a threshold, we set similarity to 0. Sparsity is related to the access density of the workload; in the workloads we used rows rarely had more than a few thousand elements even though we had tens of thousands of unique data blocks. A larger window can detect groupings that contain more elements and also have stronger intra-group similarity, but increasing the size of the window quickly meets diminishing returns [123]. The windows overlap by twice the current average group size to limit over counting. The choice of overlap is based on desired group size and is independent of the data.

For each window, the partitioning steps are:

1. Collect data
2. Calculate the pairwise distance matrix
3. Calculate the neighborhood threshold and detect groups in the I/O stream
4. Combine the new grouping with any prior groupings

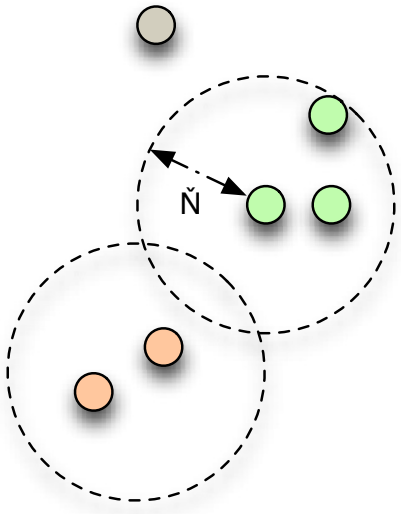


Figure 4.2: Each incoming access is compared to the preceding access to determine whether it falls within the neighborhood ($\check{N}$) to be in the same group. If it does not, a new group is formed with the incoming access.

In this technique, we start with a set of accesses ordered by timestamp. We first calculate a value for the neighborhood threshold, $\check{N}$. In the online case, $\check{N}$ must be selected *a priori* from a small set of training data and then re-calculated once enough data has entered the system to smooth out any cyclic spikes. The amount of data you need depends on what is considered a normal span of activity for the workload. In the static case, $\check{N}$ is global and calculated as a weighting parameter times the standard deviation of the accesses, assuming the accesses are uniformly distributed over time. Once the threshold is calculated, the algorithm looks at every access in turn. The first access starts as a member of group g_1. If the next access occurs within $\check{N}$, the next access is placed into group g_1, otherwise, it is placed into a new group g_2, and so on. Figure 4.2 illustrates a simple case.

Neighborhood partitioning is especially well-suited to rapidly changing usage patterns because it operates on accesses instead of offsets. When an offset occurs again in the trace, it is evaluated again, with no memory of the previous occurrence. This is also the largest disadvantage of this technique: most of the valuable information in block

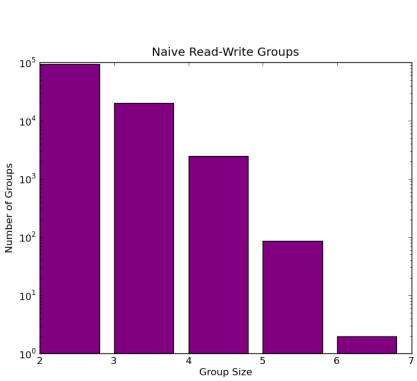
I/O traces lies in repeated correlations between accesses. The groups that result from neighborhood partitioning are by design myopic and will miss any trend data.

Neighborhood partitioning runs in $O(n)$ since it only needs to pass through the access space twice: once to calculate the neighborhood threshold and again to collect the working sets. This makes it an attractive grouping mechanism for workloads with high IOPs (for example, the XIV system from IBM can support over 300,000 IOPs), where an $n \times n$ comparison is prohibitive. Additionally, we can capture groups in real time and quickly take advantage of correlations. We also can easily influence the average group size by weighting the threshold value.

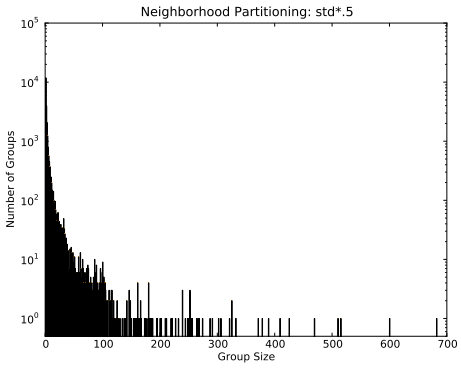
We tested the neighborhood partitioning algorithm on various data sets to get some visibility into what groupings were present in the data and whether it would be worthwhile to run our more computationally expensive algorithms. For our write-heavy, research dataset, neighborhood partitioning ended up being very susceptible to small fluctuations of its initial parameters and to the spike of writes in our workload. For the high IOPs data set, neighborhood partitioning produced a consistent grouping across more parameter values.

Figure 4.3 shows the working sets the algorithm returned with the neighborhood set to half a standard deviation, calculated over the entire trace. The read-write workload has a significantly tighter grouping because the prevalence of the writes in the cache area overtook any effect of the reads. Isolating the reads, we see in Figure 4.3(b) that the working sets become larger and more prevalent. This is due to the reduction in noise, leading to stronger relative relationships between the points that are left. We also notice that this technique is very fragile to the choice of neighborhood. For example, reducing the neighborhood to a quarter of a standard deviation (Figure 4.3(c)) causes the number of large groups to fall sharply and correspondingly increases the prevalence of small groups.

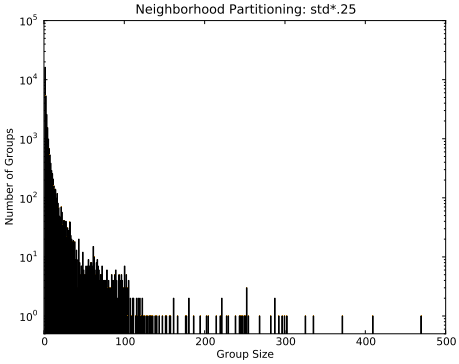
Neighborhood partitioning runs in $O(n)$ since it only needs to pass through each neighborhood twice: once to calculate the neighborhood threshold and again to collect the working sets. This makes it an attractive grouping mechanism for workloads with



(a) All Accesses



(b) Read-Only: $\sigma^2 * .5$



(c) Read-Only: $\sigma^2 * .25$

Figure 4.3: Working sets with Neighborhood Partitioning in **msr**. Groupings vary drastically based on neighborhood size and workload density.

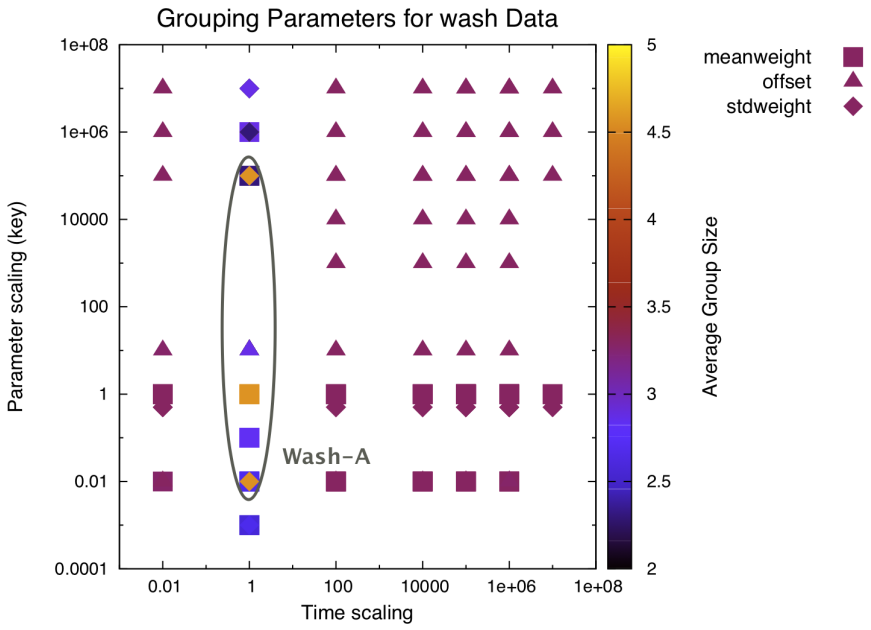


Figure 4.4: Parameter selection for the Washington data set with Neighborhood Partitioning. Color indicates the average group size produced by the specified parameters. Most parameter combinations result in grouping with the same average group size: a bit over 3. All of the datasets we tested showed similar very distinct clusters of group sizes under a parameter search.

high IOPS (for example, the enterprise system we worked with can support 300,000 IOPS, though we saw far fewer in our trace), where a full $O(n^2)$ comparison is prohibitive. Additionally, we can capture groups in real time and quickly take advantage of correlations. We also can easily influence the average group size by weighting the threshold value. As we see in Figure 4.4, we can somewhat bias our grouping parameters towards smaller or larger groups depending on the grouping application requirements. While larger groups improve prediction immediately after groupings are calculated, over time larger groups need more re-calculation to prevent false negatives as the workload shifts, potentially negating their short term predictive benefit.

4.1.4.2 Grouping Scalably: N-Neighborhood Partitioning

Though neighborhood partitioning is very efficient and has some ability to detect interleaved groupings, it does not scale well. To support arbitrarily large amounts of data, we introduce N-Neighborhood Partitioning (NNP), which merges several groupings from different windows of neighborhood partitioning without the memory overhead of a single large partitioning. By aggregating incoming accesses into regions of fixed size, NNP is highly scalable and able to perform in real time even in systems with high IOPS. The size of regions is determined by the memory capabilities of the system calculating the working sets, though increasing the size of the region quickly meets diminishing returns [123]. The regions in our implementation also overlap by a small number of accesses to account for groups that straddle the arbitrary breakpoints in our region selection. The choice of overlap is based on desired group size and is independent of the data.

As accesses enter the system, they are divided into regions and a grouping is calculated for each region using neighborhood partitioning. A grouping G_i is a set of groups $g_1, \ldots, g_l$ that were calculated from the i^{th} region of accesses. Unlike NP, NNP is not memoryless; older groupings are combined with newer to form an aggregate grouping that is representative of trends over a longer period of time.

We combine groupings through fuzzy set intersection between groupings and symmetric difference between groups within the groupings. So, for groupings $G_1, G_2, \ldots G_z$, the total grouping G is :

$$G = (G_i \cap G_j) \cup (G_i \Delta_g G_j) \quad \forall i, j \quad 1 \leq i, j \leq z$$

where Δ_g, the groupwise symmetric difference, is defined as every group that is not in $G_i \cap G_j$ and also shares no members with a group in $G_i \cap G_j$. For example, for two group lists $G_1 = [(x_1, x_4, x_7), (x_1, x_5), (x_8, x_7)]$ and $G_2 = [(x_1, x_3, x_7), (x_1, x_5), (x_2, x_9)]$, the resulting grouping would be $G_1 \cap G_2 = (x_1, x_5) \cup G_1 \Delta_g G_2 = (x_2, x_9)$, yielding a grouping of $[(x_1, x_5), (x_2, x_9)]$. (x_1, x_4, x_7), (x_1, x_3, x_7), and (x_8, x_7) were excluded because they share some members but not all.

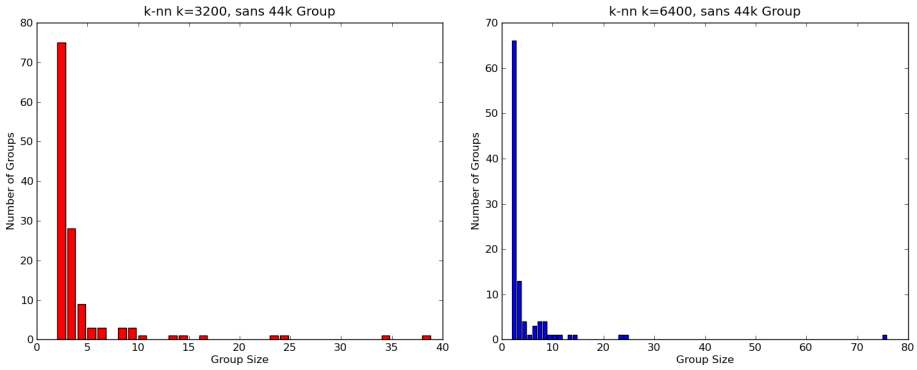
NNP is especially well suited to rapidly changing usage patterns because individual regions do not share information until the group combination stage. Combining the regions into a single grouping helps mitigate the disadvantage of losing the information of repeated correlations between accesses without additional bias. The groups that result from NNP are by design myopic and will ignore long-term trend data, reducing the impact of spatial locality shifts over time.

4.1.4.3 Nearest Neighbor Search

k-nearest-neighbor (k-NN) is based on a standard machine learning technique that relies on the identification of neighborhoods where the probability of group similarity is highest [30]. In the canonical case, a new element is compared to a large set of previously labeled examples using a distance metric defined over all elements. The new element is then classified into the largest group that falls within the prescribed neighborhood. This is in contrast to neighborhood partitioning where everything within a neighborhood is in the same group.

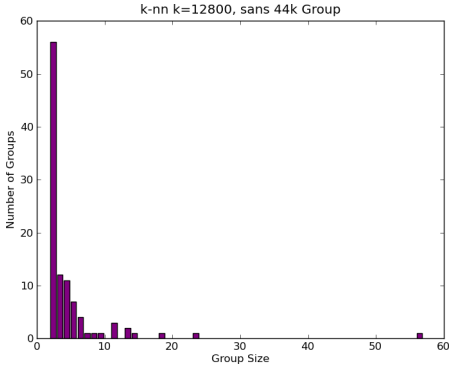
For this work, we modified the basic k-NN algorithm to be unsupervised since there is no ground truth labeling possible for groups. We also incorporated weights. The goal of weighting is to lessen the impact of access to offsets that occur frequently and independently of other accesses. In particular, in the absence of weights it is likely that a workload with an on-disk cache would return a single group, where every element has been classified into the cache group. Similar effects occur with a background process doing periodic disk accesses.

In our algorithm, we start with an $m \times m$ distance matrix as defined in Section 4.1.3 where m is the number of unique block offset values. We chose k by taking the average distance between offsets in our dataset and multiplying it by a weighting factor. For the first offset, we label all of the offsets within k of that offset into a group. For subsequent offsets, we scan the elements within k of our offset and place our offset in the best represented group. The value of k is the most important parameter in our weighted k-NN algorithm. If the workload consists of cleanly separable groups, it should be easier

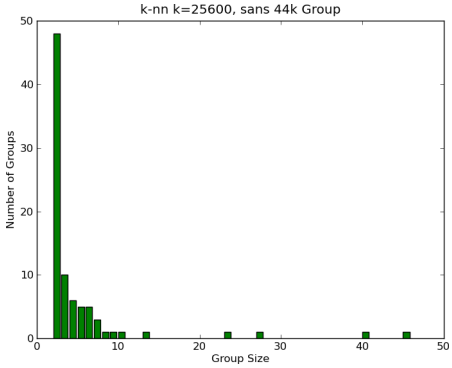


(a) k = 3200

(b) k = 6400



(c) k = 12800



(d) k = 25600

Figure 4.5: Working sets with k-Nearest Neighbor. If k is very high or low, fewer large groups are found.

to see groupings with smaller values of k. On the other hand, a small value of k can place too much weight on accesses that turn out to be noise. Noisy workloads reduce the accuracy of k-NN because with a large k, the groups frequently end up too large to be useful. We found that as long as we start above the average distance, the weighting factor on k did not have a large influence until it got to be large enough to cover most of the dataset.

Figure 4.5 shows the working sets returned by running k-NN with k ranging from 3200 to 25600. The results for the k-NN working sets are more in line with expectations, with many more small groups and a few scattered large groups. This is close to the group size distribution that NNP gives us. The groups are fairly consistent across variation,

with the larger neighborhoods resulting in somewhat fewer small groups compared to the smaller neighborhoods. Note that the graphs in Figure 4.5 are calculated after a large group representing NTFS metadata is taken out. The metadata group is a group of size 44000 that was consistently identified by both the k-NN and bag-of-edges algorithms. The consistent identification of this group is a strong indicator of the validity of our grouping techniques. For the sake of these graphs, however, removing it increases the visibility of the other groups and better highlights the differences between the variations in grouping parameters.

4.1.4.4 Graph Covering

The next method we used begins with representing accesses as nodes in a graph and edges as the distance between nodes. Presenting this information as a graph exposes the interrelationships between data, but can result in a thick tangle of edges. A large, fully connected graph is of little use, so we determined a threshold of similarity beyond which the nodes no longer qualify as connected. This simplifies our graph and lowers our runtime, but more importantly removing obviously weak connections allows us to identify groups based on the edges that remain connected. This does not impact classification since these edges connect nodes that by definition bear little similarity to each other. Once we have this graph, we define a group as all sets of nodes such that every node in the set has an edge to every other node in the set; this is defined as a clique in graph theory. Figure 4.6 shows an example clique covering of an access graph. Note that every element is a member of a single working set which corresponds to the largest of the potential cliques it is a member of. The problem then of finding all such sets reduces to the problem of clique cover, which is known to be NP-complete and difficult to approximate in the general case [21]. This is in direct contrast to nearest neighbor search, which is $O(n^2 \log(n))$.

Though clique cover is difficult to approximate, it is much faster to compute in workloads with many small groups and relatively few larger groups. We begin by taking all the pairs in a k-level distance list and comparing them against the larger data set to find

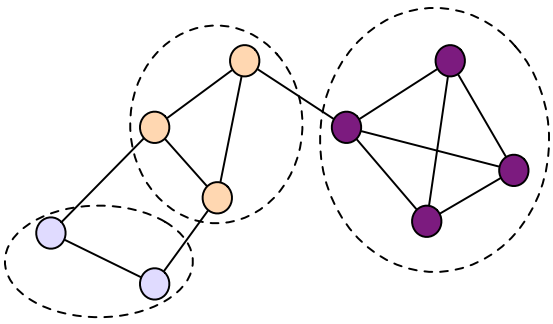
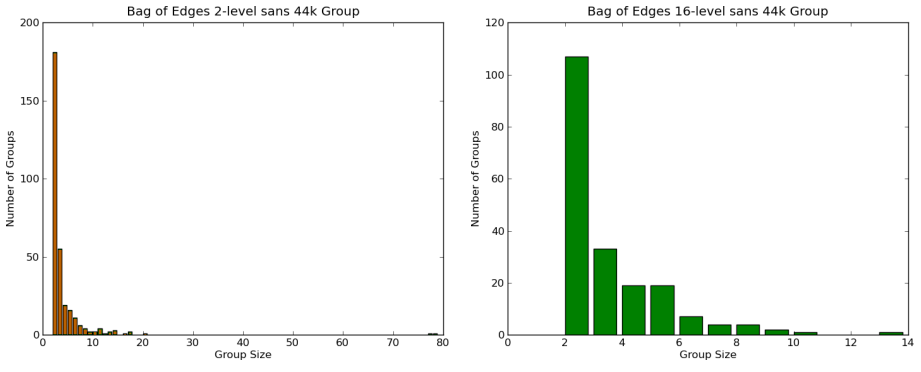


Figure 4.6: A clique cover of a graph of accesses. Nodes represent accesses while edges represent at least a threshold level of similarity between edges.

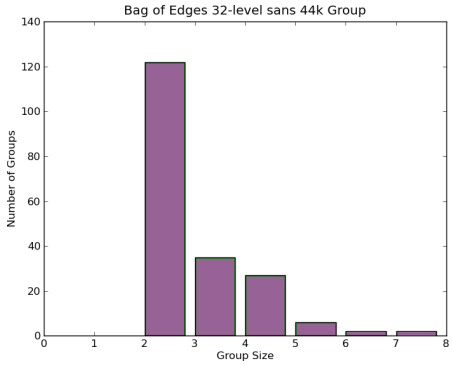
all groups of size 3. This is by far the most time-intensive step, running in $O(n^2)$. We then proceed to compare groups of size 3 for overlap, and then groups of size 4, etc., taking advantage of the fact that a fully connected graph K_n is composed of two graphs $K_n - 1$ plus a connecting edge to reduce our search space significantly. As a result, even though the worst case for our algorithm is $O(n^{|}G_x|)$ (in addition to the distance list calculation), where n is the number of nodes and $|G^x|$ the size of the maximal group, but our real runtime is closer to $\sum_2^x |G_i|^i$ where $|G_i|$ is the number of groups of size i.

We discovered that in typical workloads, this method is too strict to discover most groups. This is likely because the accesses within a working set are the result of an ordered process. This implies that while the accesses will likely occur within a given range, the first and last access in the set may look unrelated without the context of the remainder of the set and thus lack an edge connecting them. We fix this by returning to an implicit assumption from the neighborhood partitioning algorithm that grouping is largely transitive. This makes intuitive sense because of the sequential nature of many patterns of accesses, such as those from an application that processes a directory of files in order.

In our transitive model, we use a more restrictive threshold to offset the tendency for intermittent noise points to group together otherwise disparate groups of points. We then calculate the minimum spanning tree of this graph and look for the longest path. We have to calculate the minimum spanning tree because longest path is NP-complete in the



(a) 2-level (b) 16-level



(c) 32-level

Figure 4.7: Working sets with the bag-of-edges algorithm. Higher levels result in much smaller groupings.

general case, but reduces to the much simpler negated shortest-path when working with a tree. This process runs in $O(n^2 \log(n))$, since the graph contains at most $e = \frac{n(n-1)}{2}$ edges and Dijkstra's shortest path algorithm runs in $O(n + e)$ before optimization [21]. We refer to this technique as the *bag-of-edges* algorithm because it is similar to picking up an edge and shaking it to see what strands are longest. Bag-of-edges is much less computationally expensive than a complete graph covering and is additionally more representative of the sequential nature of many application disk accesses than our previous graph algorithm. We found that in our small, mixed-application workload that this technique offered the best combination of accuracy and performance.

In the **msr** data, our clique-based graph algorithm failed to ever find a group above

size two. This is useless for actually grouping data on a system since the potential benefit to prefetching one element is much smaller than the cost of doing the partitioning, and it implies that the grouping is massively overfitting. The small groups are a result of the strong requirements for being in a group that this algorithm requires: namely that every member in the group be strongly related to every other member. What this tells us is that transitivity matters for grouping, *i.e.* groups are a set of accesses that occur sequentially.

Running the bag-of-edges algorithm on this data supports this hypothesis. This algorithm is built with sequentially related groups in mind, and it returned groupings comparable to k-NN in a fraction of the time. Figure 4.7 shows the groupings bag-of-edges returns. The levels in Figure 4.7 represent the levels for the n-level distance metric, where larger levels are equivalent to more lax thresholds. The majority of the groupings are similar to k-NN, though at higher levels of distance we lose the larger groups. This is due to the lack of cohesion in large groups versus smaller ones. Though we produce larger groups, they are significantly smaller than those produced by other methods and are likely still overfitted. We also suspect that there is noise interfering with the data when the search window is too large, similar to the read-write case for neighborhood partitioning.

4.1.5 Group Likelihood and Predictivity

In even the scalable implementations, group re-calculation happens in the background during periods of low activity. As accesses come in, however, we need to update groups to reflect a changing reality. We do this by storing a likelihood value for every group. This numerical value starts as the median intergroup distance value and is incremented when the grouping is pulled into cache and successfully predicts a future access. The algorithm is structured to reinforce prior good behavior.

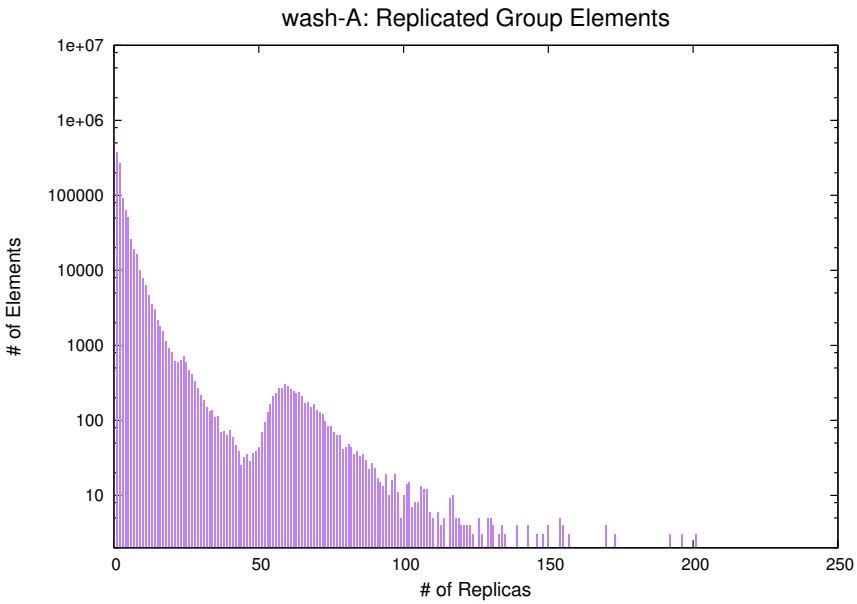
Another point where a likelihood value is necessary is when an element is a member of multiple groups. Grouping is neither $1-1$ nor onto, and it is unsurprising that one element can be in several groups. Figure 4.8 shows replicated elements in two

statistical groupings. In both, we see that most elements are members of relatively few (<10) groups. Depending on the application, elements can be indexed as members of several groups, just stored once with their most likely group, or even replicated per group instance. The elements in the long tail that belong to many groups are another indicator of the quality of the grouping method for the particular workload: more ultra-popular elements – elements that fall outside two standard deviations of the mean replica count – indicate that the grouping is overclassifing elements instead of labeling them as "noise."

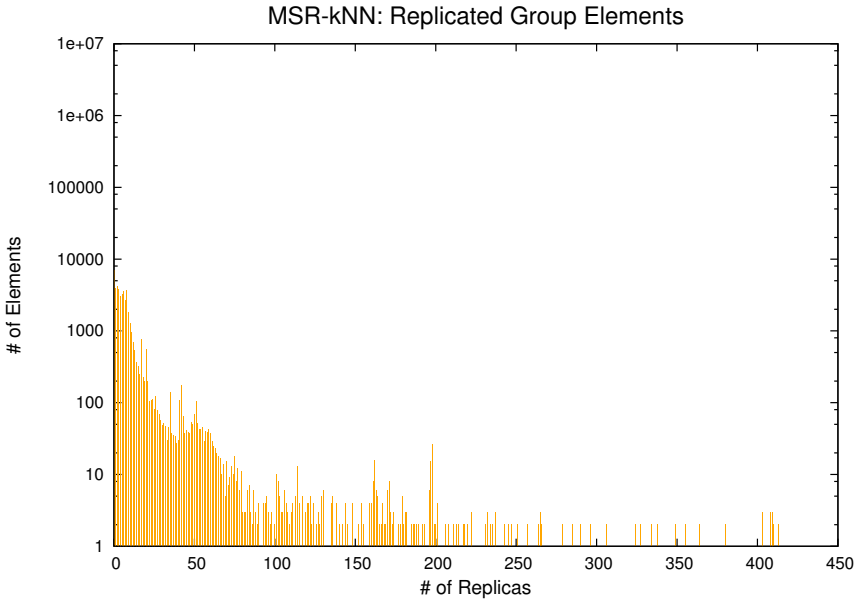
Managing likelihood and multiple replicas of group elements is handled by application domain. For instance, in NNP if an access appears in multiple groups, only the group with the highest likelihood is returned. Likelihood, which is at least initially the normalized inter-element distance within a group, is inversely correlated with group size, as we see in Figure 4.9. This serves to bias NNP towards small groupings, which we have found to have a higher average likelihood. This is expected because with fewer group members, there is less chance of a group member being only loosely correlated with the remainder of the group, bringing the entire group likelihood down.

Working sets arise organically from how users and applications interact with the data. Consequently, there is no "correct" labeling of accesses to compare our results to. Instead, we initially focused on self-consistency and stability under parameter variation. The working sets found by using the graph technique (or k-NN) are relatively stable under parameter variation as long as the search space for determining distance between access points remains fixed. We expect there to be variation here as a result of natural usage shifts or cyclic usage patterns. We have obtained a limited number of data sets that contain extra parameters such as initiator ID and process ID that could corroborate groupings. Using this data, we verified that stable groups in these data sets are also products of the same process or initiator more than 90% of the time.

We expect groups to be interleaved, so we can not just check for the same groups to occur in many traces. Instead, we use the Rand index [86], a method to compare clusterings based on the Rand criterion. The Rand index is a good comparator for groupings because it is essentially a pairwise similarity comparison across groupings, meaning

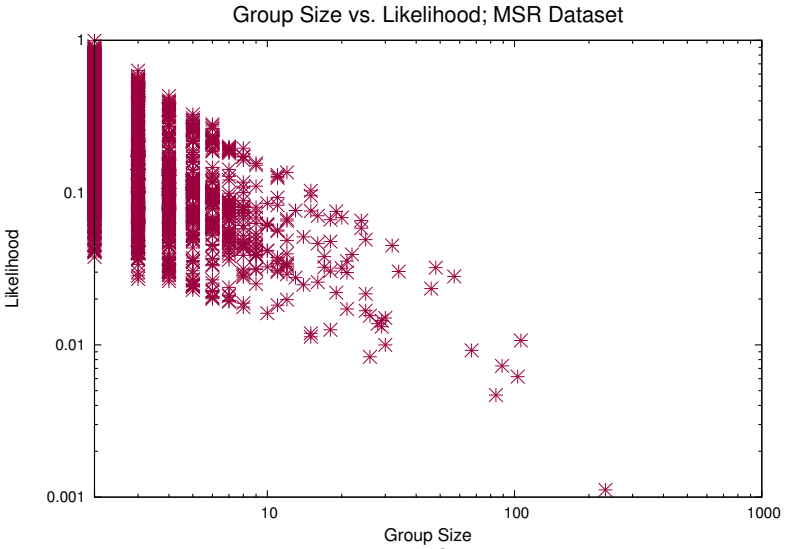


(a) `wash-A` with NNP

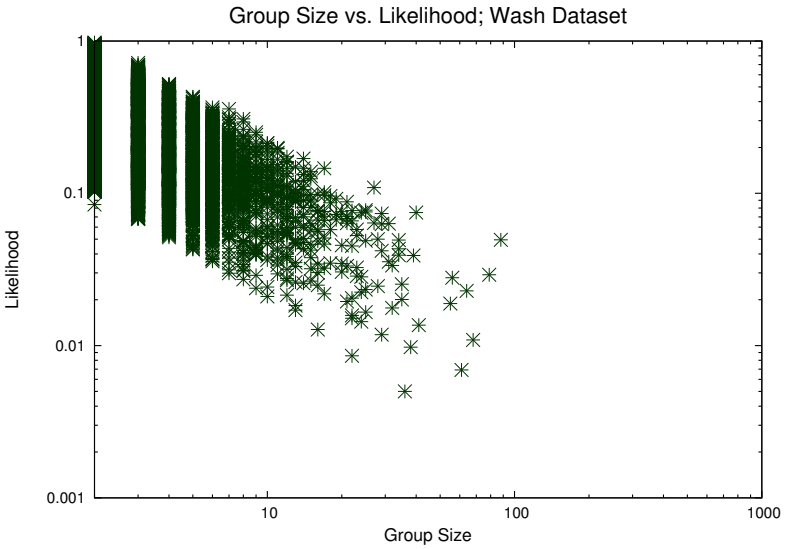


(b) **msr** with k-NN

Figure 4.8: Replicated elements for two statistical groupings



(a) **msr** with k-NN



(b) **wash** with NNP

Figure 4.9: In both NNP and k-NN, the most likely groupings tended to be smaller across all of our data sets. These graphs represent a subsample of 10% of groups per grouping, randomly selected. Likelihood is normalized within each grouping.

that it does not penalize groups for the type of small changes that we expect to see in our groupings from expected usage shift and lack of complete data. For example, if a group of four elements adds in a fifth member during the testing trace, the group is still considered to be a correct grouping. The Rand index between groupings G_1 and G_2 is calculated as

$$R(G_1, G_2) = \frac{a+b}{\binom{n}{2}}, 0 \leq R \leq 1$$

where a is the number of pairs present in both G_1 and G_2, b is the number of pairs present in G_1 but not in G_2, and n is the total number of possible elements in any grouping.

We can also calculate mutual information between groupings in order to gauge grouping consistency over time. Mutual information is an information theoretic measure of the independence of two sets relative to each other. For groupings G_1 and G_2 with marginal entropy values $H(G_1)$ and $H(G_2)$ respectively, the mutual information is defined as $H(G_1) + H(G_2) - H(G_1, G_2)$. While the Rand index is a strict pairwise comparison, mutual information should allow groupings to be looser and still be considered "good." We should be able to test both of these methods simultaneously to decide what is actually a better measure for tuning groupings.

Depending on the expected use of grouped data, the best test will be to run a large enough data set and track how often the grouping provided a good result on new data. At that point, the grouping can even be quickly modified in place based on how often the grouping does or does not improve the desired performance in terms of disk spins, cache hits, or other measures that provide observable benefits.

On larger data sets, we can do actual verification on the predictive power of our grouping selections. While we were implementing grouping for both HANDS (Chapter 7) and PERSES (Chapter 6), we tracked the predictivity of groups over time. Figure 4.10 shows the predictive power of a grouping for the **ent-storage** dataset. We see that the predictive capability decreases after a certain point, but is fixed by re-grouping based on recent accesses.

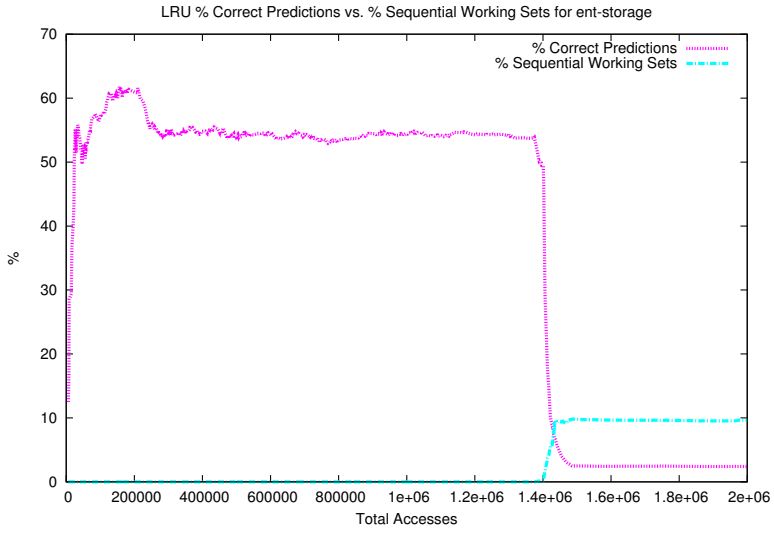


Figure 4.10: Predictivity over time for NNP groups from the **ent-storage** data set.

4.1.6 Clustering

Clustering refers to a class of unsupervised learning techniques that group $n-$ dimensional heterogeneous data. Many techniques rely on a known or predictable underlying distribution in the dataset or a derivable number of clusters. Additionally, many clustering methods need to perform expensive computation to add points. We have found that popular methods such as k-means give undue weight to very large groups and ignore smaller or partially overlapping clusters. This is especially unfortunate as we believe the actual grouping underlying all of the data sets we have observed thus far is strongly biased towards small clusters, and small clusters are better for applications such as pulling data into cache. These properties make many clustering algorithms unsuitable for real-time grouping selection.

One class of clustering algorithms that show promise are agglomerative. In agglomerative clusterings, every element starts as its own cluster and the clusters are then merged until the algorithm converges. iClust is a particularly strong candidate: it does not require an *a priori* similarity metric or underlying distribution, it is invariant to changes in the representation of the data, and it naturally captures non-linear rela-

tions [102]. iClust has been used mainly in biology applications to cluster genes by expression data such that members of a cluster have high co-dependency [131]. Though we did not use iClust to calculate the groupings for our applications because it is still much slower than NNP, we believe that for a large, static system it would outperform bag-of-edges on more volatile data.

4.1.7 Analysis

The next several chapters show the predictive power of the groupings we find in the data in different applications despite the sparsity of the traces. This indicates that it is worthwhile to look for groupings in similar multi-user and multi-application workloads even if the only data available is block offsets and timestamps. Our group consistency results (Section 4.1.5) indicate these groups are representative of real-world phenomena and thus should have a high predictive value. Once we can run a supervised study on larger data, we should be able to confirm this relationship. Being able to collect useful workloads without impacting privacy or performance is invaluable for continuing research in predictive data grouping. This also reduces the cost of our analysis substantially, since we can determine whether a workload will be separable before trying advanced techniques to identify groupings and disrupting the system to group working sets together on disk.

A concern early on was that the access groups we discovered would overlap, leading to a need to disambiguate and manually tune our models to the data set. It turned out, however, that in every classification scheme we used we never had overlapping group chains. This allowed us to keep our methodology general and more likely to be easily portable to other data. More importantly, this is a strong indication that our groupings represent separate access patterns. If they did not, it is likely some of them would have had over-lapping components since the accesses are uniformly distributed.

We realize the assumption that block offsets do not uniquely identify a block is not strictly true in some systems. A majority of data that is frequently overwritten is in the cache block, however, and this is consistently identified as a single group by our

algorithms. The less frequent overwrites that occur as a result of normal system use should be handled by the adaptability of our algorithms over time. If the content of a block offset changes, it will start being placed into different groups as the algorithms update the distance matrix.

4.2 Categorical Grouping

Categorical grouping by definition requires some functional knowledge of the data which relies on human curation, either in manual labeling or metadata upkeep [2]. Example attributes include metadata such as size, name, type, owner, path, or even whole file content. Additionally, sometimes it is possible to have higher confidence priors by manually grouping data according to projected, controlled use. We do not attempt to re-create work here. We have used frequent sequence matching on file contents to automatically extract a group from a cache of water quality logs and we have used pre-defined labels in two datasets to test pre-labeled, categorical groupings

4.2.1 Metadata-based Grouping

With rich metadata, we could partition data with more domain specific awareness of how it is accessed by users and applications. There are two directions this could go: popularity modeling or metadata partitioning.

If we have access to workloads with metadata, we could extend the work of Parker-Wood *et al.* on metadata partitioning for security [79]. This would allow us to express our data in terms of trees of related components that ought to have a high probability of codependent access.

The other technique we are interested in comes from the PDC work [80] and involves using Zipf's law to characterize workloads based on their content. Zipf's law describes a power law distribution that was first introduced by George Zipf to describe the relative frequencies of certain words appearing in a language corpus [50]. Several authors [13, 105] have shown that Internet workloads have a Zipf-like distribution with either one or

two modes. The modality is dependent on the type of delivery system, with streaming systems tending to be bimodal [105]. In the absence of archival workload data, we are assuming a web-like workload and using a unimodal Zipf-like distribution to represent popularity of files. We expect our archival by accident workloads to have some of the same properties as the web (flash-crowds, in particular). Over a large enough set, one can even argue that the WWW is inherently an archival system. The top 10 websites accounted for 40% of page views in 2006, and there is evidence that the drop off is exponential instead of a long tail [68, 116].

A "Zipf-like" distribution is specified by Ω/i^{α}, where α corresponds to hotspot size and Ω is a normalizing constant. The cumulative probability that one of the top k pages are accessed from a set of N pages is $\phi(k) = \sum_{i=1}^{k} \Omega/i^{\alpha}$. Ω can be approximated as $(1-\alpha)/N^{1-\alpha}$. This sort of analytical grouping model could prove stronger than the data mining approaches we have taken so far.

4.3 Summary

In this chapter, we have discussed two types of grouping, statistical and categorical, and how we implement both of these. We will show more instances of these groupings on different datasets in the next three chapters.

Statistical groups are found by tracing accesses, and then examining correlations between data within a certain temporal window of accesses to derive a grouping. This grouping is frequently combined with groupings from previous temporal windows to get the best classification for the data, though there are highly dynamic workloads where a myopic grouping may outperform one that includes long term data. Fast partitioning algorithms such as NNP are best for rapidly changing data where groupings are likely to shift quickly, whereas a clique-based technique such as bag-of-edges is best for archival-like data where limited data movement is a greater concern and groups are less likely to shift. Categorical grouping, if well maintained, is a good solution for grouping a particular storage system, but guaranteeing metadata accuracy is difficult, and these

groupings are relatively immutable.

Combining these two techniques is a natural progression of this work, and we believe, when the appropriate data is available, that combining these techniques will give categorical groupings the flexibility and tolerance to misclassification that they lack.

Chapter 5

Grouping for Power Management

If you want to find the secrets of the universe, think in terms of energy, frequency and vibration.

– Nikola Tesla

One of the most pressing needs of any large data storage system is power management. This includes both power to the machines and power to the cooling systems required to keep the machines from overheating. Previous experiments have shown that the hard disk is the largest contributor to power as well as a primary source of heat in many common configurations [81]. Power management for storage systems boils down to reducing disk activity as much as possible. This is a natural application for grouping because data that is accessed in proximity requires much less disk movement. As a result, primary motivation for investigating grouping data on large systems is the growing need to save power on these large, high-temperature storage clusters. There are several strategies to saving power, as discussed in Chapter 2, Section 2.3.1. We propose that any method that works on large, archival-like data stores will by necessity need to be agnostic of the workload of the system. Additionally, disk drives are likely to persist as an inexpensive large scale storage medium even as a variety of storage class memories come into market [55]. Previous efforts to conserve power by spinning down idle disks failed because of the high cost of disk spin-up. We propose that grouping data in data centers and spinning up groups of disks for a period of time after any element

from that group is accessed will lead to an overall reduction in power consumption when amortized over time.

Research in archival systems, as we discussed in Chapter 2.2.1,has typically assumed a "write-once, read-maybe" workload. Real traces are hard to come by, so a significant body of work exists using simulators that assume that reads and overwrites are relatively rare. Real workloads, however, may have significant numbers of reads and overwrites in an area of the system that is "hot," *e.g.* the NTFS master file table is read constantly and overwritten whenever files are altered. Many workloads that are considered archival may be susceptible to a quick change in status from archival to active due to a variety of reasons including periodic audits, a current event, or a renewed research interest. This bursty activity may be disproportionately costly in archival systems that assume most disks are idle most of the time.

5.1 Power Savings in a Pre-grouped System

We approach this problem by using categorical tagging to store similar data across the same sets of devices. We presume we have a MAID-like storage system where disks are spun down when not in use, and they are spun up on access and left on for a period of time after an access is served. Our goal is to group data to reduce the percentage of accesses that result in disk spin-ups and thus increase power efficiency and reliability in a more realistic archival system. We use both given and automated classifications to create our groups and find that if the group is left spinning for 50 seconds after a spin-up, enough subsequent reads are caught by the spinning groups that the power cost is up to 52% lower than the alternatives of spinning up for every read or always leaving disks spinning. We chose 50 seconds based on work in mobile disk spindown [43, 28], and subsequent experimental testing showed 50 seconds to be locally optimal for power savings.

Our work is focused towards systems that have a large number of physical disks. With fewer disks, there is a possibility that leaving the disks spinning continuously

would out-perform group-based selective spin-down. There is also an additional overhead of an index server to handle the mapping between blocks and groups. Multiple, hierarchical index servers may be required in a larger system. OSDs such as Pergamum are ideal because they have the added ability to use on-board NVRAM for read and write caching, allowing for a smaller main index and msr-traces [108].

After an initial setup period, we categorically split our data into groups. These groups could be formed around a variety of semantic or incidental labels such as timestamp, filesystem placement, writer, filetype, the authors in a LaTeX document, etc. These groups may be dynamic, so files must be able to move as access patterns change. Chapter 4.1 goes into more detail of how these groups are determined. Access groups do not share hardware, and when any data is read from an group the entire group is spun up and left on for a set amount of time. te

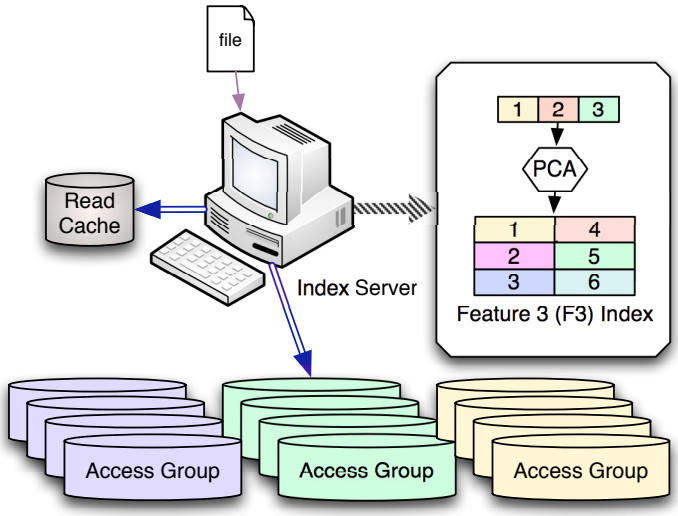


Figure 5.1: Typical system components: disks, cache, and index server

There are a few additional, common storage system features that benefit from this sort of power management. One obvious concern is that a flash crowd could focus on just one element of an group, leaving the rest of the group spinning for no benefit. This issue is easily handled with a modest read cache, such as the LRU cache PDC uses [80]. While this is less of an issue in a classic archival workload, a modern archival or archival-by-

accident workload needs a level of caching to soften the impact of an influx of accesses to a data set caused by anything from a news event to a unexpected mention in a major blog.

Figure 5.1 shows the design of a basic system that uses groupings. The index server stores the mapping between files and access groups and calculates the appropriate group for an incoming file. If these groups are stored on OSDs, the index server can treat groups as atomic entities and rely on the OSDs to index files within the group. When data is read, the data is first checked for in the cache. If there is not a cache hit, the index server identifies the group that the indexed copy of the data is on and sends the read to the appropriate group of disks in addition to copying it into the cache.

Data to be written is sent to the index server. The index server classifies the incoming file or files using techniques such as support vector machines or component analysis to chose the appropriate groups for the files [30]. If a file group runs out of disk space, either disks are added or the file group is split. The standard approach is to assume that archival systems have enough excess capacity that empty disks will typically be available [127, 56]. However, we have found that smaller groups have better power savings outcomes. Small groups also make better use of hardware.

We analyzed two static file access traces to determine the magnitude of power savings to be gained from pursuing this idea, the **water** and **wash** datasets described in Chapter 3. All analysis in this work is at the file level. As a reminder, the **water** data consists of 90,000 queries to a record store from 2007 through 2009. We make the assumption that queries correspond to record accesses. The data set is pre-grouped, and the grouping labels we consider for each access are "Timestamp", "Site" "Site Type", and "District."

Though many group classification schemes are valid for our data, we chose to use a pre-defined partitioning for our experiments to focus on the potential for power savings. We were able to automatically group the accesses according to the "Site" metadata field by indexing the whole text by word and doing a component analysis, but this classification was identical to the "Site" classification provided in the pre-defined labeling.

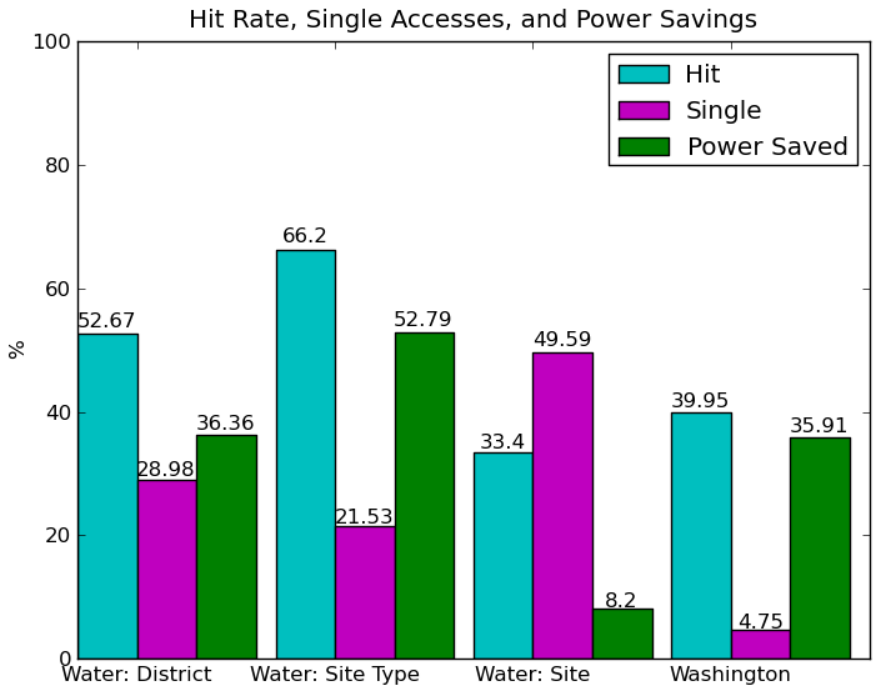
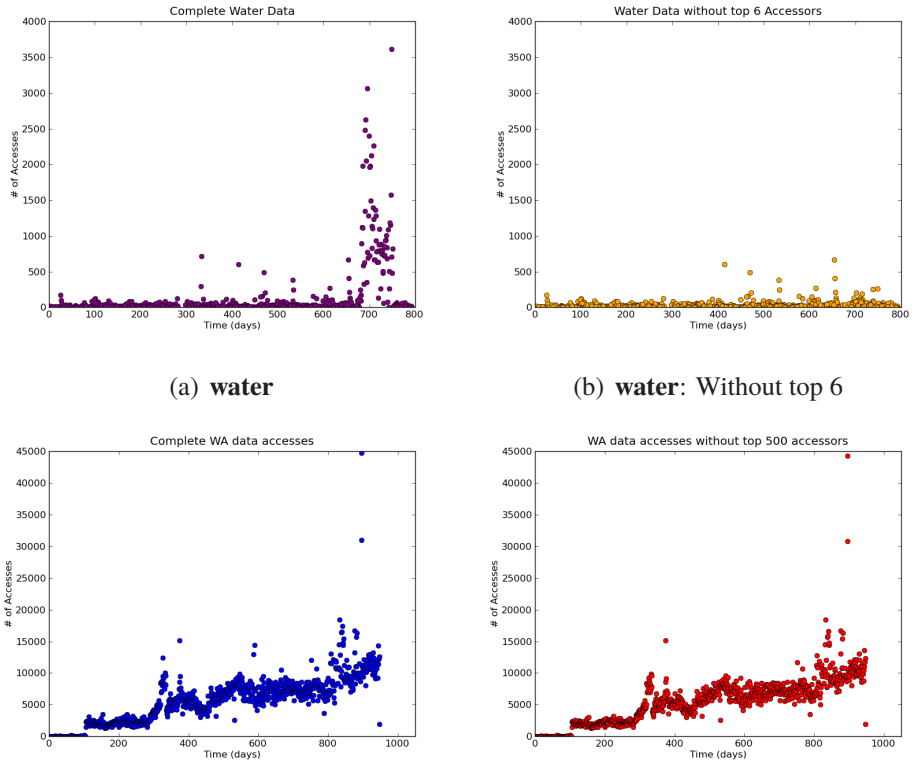


Figure 5.2: Percentage of accesses that hit a spinning file group (Hit), the percent that caused a full spin period without a subsequent hit (Single), and the power savings compared to spinning up the disk on demand (Power Saved) for various classifications. Percents are compared against a random allocation.



(a) **water**

(b) **water**: Without top 6

(c) **wash**

(d) **wash**: Without top 500

Figure 5.3: Access patterns for the **water** and **wash** datasets with and without top accessors

wash is database of vital records from Washington state where records are labeled with one of many type identifiers (*e.g.* "Birth Records", "Marriage Records"). We examined 5,000,000 accesses from 2007 through 2010 to a 16.5 TB database. Our goal was to see if categorical data placement increases power efficiency in a system where disks are in a low-power mode or powered off entirely much of the time.

Outside of the minimal overhead of the index servers, we can do a quick calculation to show the value of working in this direction. Suppose a spin-up costs 100J while leaving the disk on costs 3J/sec ([25]) and disks are left on 50s after an access in hopes of catching another access to the same group. In a system without any groupings, 100 accesses would cost:

$$p = 100 \times (100 + (3 \times 50)) = 25000\ J$$

In a system where even 30% of these accesses hit an already on filegroup, the power requirement falls to

$$p \leq [70 \times (100 + (3 \times 50)) + 30 \times (3 \times 50)] \leq 22000\ J$$

The inequality is necessary because we do not know how far into the 50s period a subsequent access occurs. This shows that 30% hit rate, which we see in our data, results in at least a 12% power savings.

5.2 Results

Figure 5.2 shows the percentage of accesses that hit a spinning file group (Hit), the percent that caused a full spin period without a subsequent hit (Single), and the power savings compared to spinning up the disk on demand. While the **water** Data Set is only 2.3 GB, the access patterns are still meaningful because they are independent of the data size. This data is also a good test because it attaches predefined groupings such as "Site Type" to each file.

If we break up the data into 7 disjoint groups based on site type and calculate the *hit rate*, defined as the number of accesses made within 50s of each other, we find that leaving our disks spinning would have covered 63.5% of accesses, as we see in Figure 5.2. Figure 5.2 also shows the impact of singles on the power savings, especially on larger groups such as "Site". This is logical since each of those is a demonstration of our worst case, where a single access costs a spin-up and a full 50s of additional active drive time for no benefit (though this time could be used for scrubbing and offloaded writes).

Figure 5.2 shows the power saved with our method versus spinning up disks as needed, leaving them on for a fixed period of time, and then powering them off. For a different perspective, Figure 5.4 shows the power savings over time compared to a system with always-on disks. Though these disks avoid spin-up costs, the batched nature of the accesses make leaving disks spinning a poor method of saving energy. The spike near Day 700 is caused by an influx of queries from a search indexer. Figure 5.5 shows the relative power savings of grouped allocation with and without the search indexers

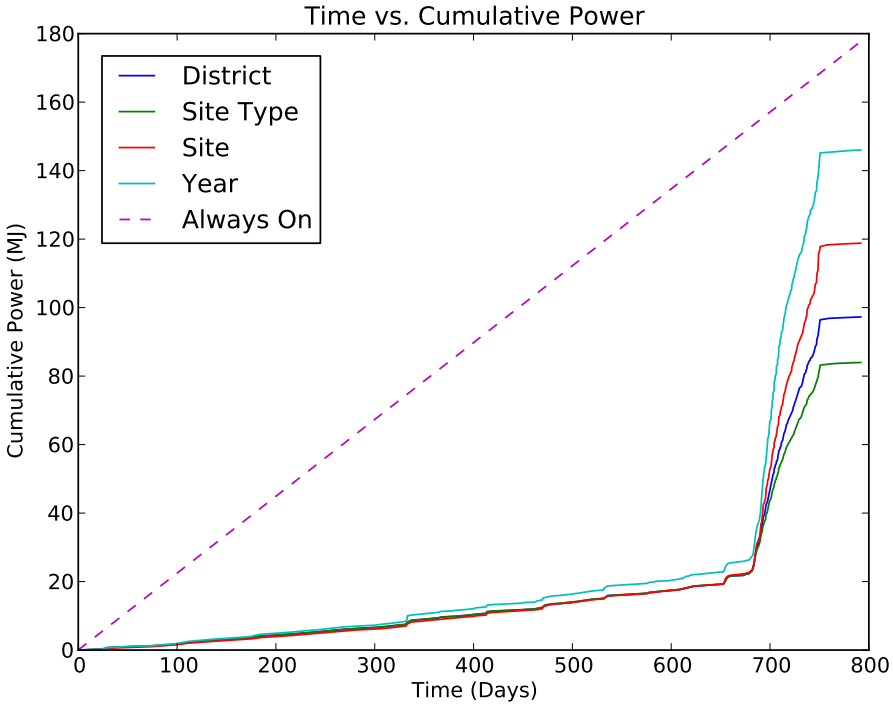


Figure 5.4: Power Savings for various groupings versus leaving disks on. *ontime*=1s, *spin-time*=50s

affecting the results. Note that two of our three groupings actually lose power savings when indexers are removed. This happens because there are so many fewer files being accessed, and the data is presented to the indexer in Site order.

5.3 Discussion

The major drawback of this approach is that, assuming all disk groups are equal, disks that do not have subsequent accesses will be left on longer than they otherwise would have been. Though 50 s worked for our system, other systems should weight this time by the power their disks consume in both idle and spinning states. This number should increase as disks gain efficiency.

Label	*# of Groups*
`District`	9
`Site Type`	7
`Site`	680

Table 5.1: Water Quality Group Sizes

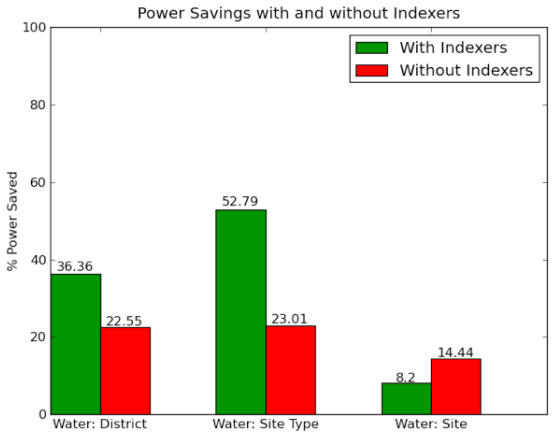


Figure 5.5: Effect of Indexers on % of Power Saved.

Though we do not explicitly model writes, we presume that writes in our archival-by-accident workloads can be offset with the cache located either on the index server or a access local group controller [75]. Writes will then be distributed among the disks in the group when the group is spinning for a read. We also recommend a read cache to handle repeated, consecutive accesses to a single file, which reduces the power footprint of a flash crowd.

As groups change, we can track their predictivity through the same reinforcement-based technique we use for statistical likelihoods (Chapter 4.1.5), and re-allocate data lazily as disks are spun up. We expect that the overall performance impact will be limited after the initial groups are determined in an archival workload. Any computation for classification could be distributed across the nodes in an OSD cluster.

Chapter 6

Grouping for Availability

For want of a nail the shoe was lost.
For want of a shoe the horse was lost.
For want of a horse the rider was lost.
For want of a rider the message was lost.
For want of a message the battle was lost.
For want of a battle the kingdom was lost.
And all for the want of a horseshoe nail.

– English Proverb

Another primary concern for a large scale storage system is the tradeoff between data reliability and availability in a hybrid storage environment.

Over time, failure events are inevitable in large disk-based storage systems [96, 7]. Even in failure events where data is not lost, there is a system-wide cost for accessing data on the failed disk. The failed disk is inaccessible, and other disks and the interconnecting network links may be saturated. This cost is increasing as growth in disk size outpaces growth in disk, CPU, and network bandwidth [55, 63]. While a 100 GB drive that can be rebuilt at 50 MB/s is fully operational in about half an hour, a terabyte drive at 50 MB/s can take up to six hours to be back at full bandwidth. Additionally, during a rebuild there is often significant network overhead that can have negative side effects on disks that are not even involved in the rebuild.

We examine the issue of excessive rebuild time from the perspective of *projects*:

groups of files that are frequently accessed together. If a failed disk contains even a small amount of critical data for multiple projects, all of those projects could suffer degraded performance until the rebuild is completed. Examples include developers losing a header file that a build depends on, a high performance system that loses an input and misses its time slot, or a cloud provider that breaks a performance SLA.

If, however, the failure occurs on a device that is not actively being accessed, it has almost no productivity cost: it is the proverbial tree fallen in a forest. PERSES is a data allocation framework designed to decrease the impact of device failures on the productivity and perceived availability of a storage system. We name our system PERSES, after the Greek titan of cleansing destruction, because it focuses destruction in a storage system to a small number of projects so that others may thrive. Our goal with PERSES is to isolate failures away from as many projects as possible.

Multi-user cloud storage systems, a primary target of PERSES, are caught in the data deluge. According to the 2011 IDC report, digital content is expected to grow to 2.7 zettabytes by the end of 2012, which would be a 48% increase over 2011 [37]. Cloud applications such as web hosting and document sharing maintain SLAs that guarantee low response time to users. In this environment, the cloud provider saves money by isolating failures to, and thus breaking SLAs with, as few customers as possible. While they experience more degradation, the marginal cost of additional slow response time is often lower for users once any slowdown has occurred.

Exacerbating this problem, many systems sacrifice rebuild speed in exchange for increased reliability. These systems typically place data evenly across disks in order to distribute and thus dilute the impact of failure events. When a failure event does necessitate a rebuild, however, the rebuild can impact a wide array of people or projects even though each could have only lost a very small amount of data. By combining existing reliability measures with selectively laying out data according to group membership, we can isolate failures so that they impact fewer users or projects, increasing the perceived availability of the system.

Other use cases that are disproportionately affected by partial data degradation in-

clude compilation and select scientific workloads. Compiling large code bases typically involve multiple dependencies, and developers working with large code bases often have to wait to compile until all of the files they need are available. An unlucky failure can halt the compilation of large swathes of code for want of one file. Similarly, scientific data analysis can rely on a small number of key files such as format descriptors or configuration files that slow the analysis of terabytes of data.

PERSES is also a good fit for large storage systems where a small but constantly shifting subset of the data is in active use. Systems that organically grow around a dynamic set of requirements naturally tend to have a small set of data in active use and a long tail of accesses across the remainder of the data [106]. These systems resemble archives in that while there are distinct, consistent projects, there is little repeat access for popularity based allocation to take advantage of. We study such archival-like workloads in this paper to form a baseline for improvement.

PERSES improves availability by laying out project groups adjacently on disk to isolate faults. Maintaining an out-of-band index of all of the data in a project requires significant administrative overhead. To avoid this, PERSES automatically calculates associations in historical trace data to identify likely project groups. Since elements of a project are co-located on disk, any single failure impacts a very small number of projects. We derive project groups from either trace metadata or access patterns. We use NNP to calculate projects in O(n), where n is the number of accesses in the training set, without the overhead of maintaining groups by hand.

To demonstrate PERSES, we injected failures in the **wash** and **water** datasets and ran them through a disk simulator with a variety of layouts based on calculated project groups. We show that layout under PERSES leads to faults that affect fewer projects during the rebuild phase than a group-agnostic layout. Our fault injection simulation compares the real time lost during disk rebuilds across layouts by measuring *project time lost*, which we define as the total delay encountered by all project groups as a result of the access requests that are delayed while disks are rebuilding.

We found that in our best case, PERSES gained over 4000 hours of group time dur-

ing the three year trace compared to a random allocation and gained over 1000 hours compared to an ideal temporal allocation. Additionally, we found that while rebuild speed was a major factor in relative project time lost, adding a read cache made no noticeable difference. Finally, we discovered that setting a minimum group size improved the performance under PERSES even on smaller disks, with a maximum productivity improvement of 94% across large groups.

Our main contributions are:

1. A methodology for project detection for better performance during failure events (PERSES).

2. A fault simulator with real trace data showing up to 80% decrease in project time lost with PERSES allocation.

3. A direct comparison of statistical and categorical project detection techniques.

4. Parameter identification and optimization strategies for rebuild speed and minimum group size.

PERSES is inspired in particular by D-GRAID, which demonstrated that placing blocks of a file adjacent to each other on a disk reduced the system impact on failure, since adjacent blocks tend to fail together [100]. Our grouping methodology will allow for failures to be localized to projects, which represent use cases, allowing more of the system to be usable in case of failure. D-GRAID proposed to analyze the reliability increase of distributing blocks of a file across as many disks as possible versus keeping the blocks together on not only the same disk, but the same track. They proposed that since a head error is likely to affect adjacent track members more than random blocks on a disk, writing a file consecutively was an effective way to localize failures and thus minimize the person or project time lost as a result of file unavailability during the rebuild process. PERSES translates this idea to cloud and other large storage arrays.

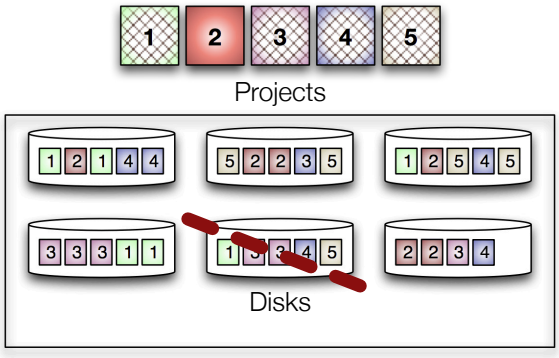
6.1 Design

PERSES is a statistical layout algorithm designed to isolate faults to as few projects as possible. The two main functions of PERSES are project detection and layout. Projects are automatically learned by extrapolating relationships based on a period of accesses and modifying these groups as new data enters the system. This unsupervised approach to project detection is a better fit for dynamic data where the meaning of curated labels drifts over time and for systems with privacy controls or performance barriers to collecting metadata-rich traces. Since we learn our projects, they may not map precisely to real applications or users. In fact, to avoid overfitting to past data, it is important that the mapping be inexact. We have found, however, in previous work that over time, our statistical groups have over 90% correspondence to specific PIDs or UIDs over the next time period [122].

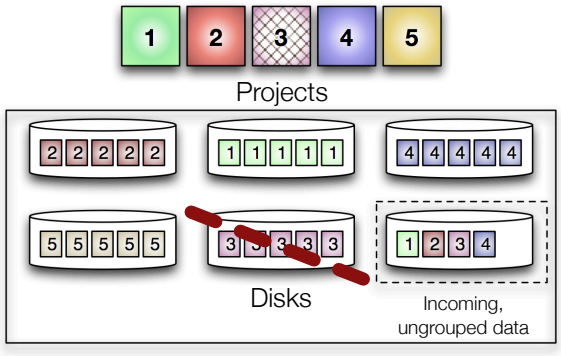
We started with the hypothesis that the particular grouping technique used is less important as long as groups are predictive. We tested two grouping techniques on each of two labeled data sets. For the metadata analysis, we selected which labels to select groups with based on prior work doing component analysis on this data [121]. For the statistical analysis, we used N-Neighborhood Partitioning (NNP), which we discussed in Section 4.1.4.2. As a reminder, NNP accumulates a window of I/Os, calculates a sparse $n \times n$ distance matrix over the window, determines groups based on density, and then updates the prior grouping based on this new information. We chose NNP because it biases towards smaller groups, reducing the possibility of cache churn, and because it runs in $O(n)$, which allows us to quickly reclassify when predictivity begins to drop.

6.1.1 Layout

Most storage systems spread out data across several disks to maximize resilience against disk failure. When a disk fails, access to data on the failed disk is degraded from having to reconstruct data from parity or backups. PERSES lays out data according to project to reduce the impact these periods of degraded access have on the users of the



(a) Standard. Single failure impacts Projects 1,3,4,and 5



(b) With PERSES. Single failure impacts Project 3

Figure 6.1: Projects are represented by numbered boxes. When a disk fails, all projects that have data on the disk have degraded performance. PERSES dynamically groups data together on disk to limit the performance impact of failures across projects.

system. The key insight is that the impact on productivity for a project can be disproportionate to the amount of data lost. For example, missing an input file to a scientific simulation can cause the simulation to stall and overrun its allocated time. Missing part of a virtual machine boot sequence can cause several users to have to wait to restore images. Missing a single small file could break the build for an entire code base.

Our layout places as much project data as possible on the same disk. This way, when a failure occurs it impacts fewer projects, which means the system is more productive. We measure this improvement in system productivity by summing the total

project time lost. The ideal disk failure is caught by a monitoring process (such as disk scrubbing) and repaired before any accesses are made to the lost data. We refer to this as an *zero-impact* failure. Intuitively, on systems with bursty access patterns, zero-impact failures become increasingly common when data has high locality. One high level goal for PERSES is to maximize the number of failures that do not impact system performance.

Figure 6.1 shows what a failure would look like on a trivial shared system with and without PERSES. The numbered boxes correspond to blocks of data, and the numbers correspond to projects. These projects could be sets of user data, large data set analyses on enterprise, scientific experiments, code bases, etc. In the ungrouped example, Figure 6.1(a), data is arranged without any consideration to project membership. When one disk is lost, indicated by the thick dashed line, up to four projects could see performance degradation if they are accessed before rebuild completes. On a system laid out with PERSES, such as Figure 6.1(b), the data is laid out across the disks based on project membership. When a disk fails in this scenario, only Project 3 is affected since it is the only project on the failed disk, both reducing the amount of degradation across groups and increasing the probability that the failure is zero-impact. Groupings are refreshed periodically and the layout re-arranged, so some data will always be unlabeled. This unlabeled data is written serially to a reserved area on disk until the next grouping cycle occurs.

We assume that the underlying system uses parity to rebuild data when a disk is lost, but we do not specify a particular reliability scheme. This is intentional since even though PERSES is designed for a parity-based reliability architecture, all reliability schemes save for local mirroring introduce a reconstruction delay. We care about reconstruction speed and consequent time degraded, but not the reliability method. Rebuild requests are typically given very low priority compared to incoming reads and writes, so a parity system or backup server should have little overhead that we do not account for [44]. In our simulation, we model a range of rebuild speeds to better understand the impact of PERSES in different reliability environments.

PERSES is designed for a system with multiple disks, different projects that rely on

many blocks of data, and a controller to group data and manage layout. We also assume the ability to collect minimal trace data to inform the grouping algorithm. The best candidate systems for PERSES have a number of data disks greater than or equal to the number of projects or working set groups on the system, though we see in Section 6.2.1 that this is not strictly necessary.

6.2 Methodology

We built a trace simulator with stochastic fault injection to analyze PERSES on a range of different hardware configurations and rebuild environments.

Our simulator goes through the following steps:

1. Initialize disks and cache
2. Determine initial groupings
3. Lay out data across disks
4. Step through a real data trace; update grouping as needed

Disks are initialized with all of the data that is read through the course of the trace. Disks are then filled progressively with either grouped data, data placed in order of trace appearance, or randomly placed data such that the only empty space is on the final disk, depending on the experiment. The amount of data each trace accesses is fixed, so as we add more disks to the simulator there is less data per disk. Since we do not know the size of the accesses in our trace, we arbitrarily allocate 10 MB per access. We retain generality since our simulation results can be translated to any system with fixed size blocks by adjusting the data to disk ratio accordingly. The simulator uses an LRU read cache to capture popular accesses. We chose LRU because it is well understood, and we did not explore further because, for our workloads, cache size had no impact on our results until the cache was over 100 GB. The default cache in the following experiments starts cold and is 10 GB. This cache is assumed to be in memory and unaffected by

disk failure. As we will see, the cache size has little impact on the end result outside of shifting the result graph.

An initial grouping is calculated before the trace is run, as described in Chapter 4. Groups are then laid out sequentially on disks starting with the smallest groups. In the ungrouped experiments, files are laid out randomly or temporally without attention to project membership. Modeling correlated failure and exploring alternate strategies for assigning groups to disks are in our future work.

Disks are initialized with a low, uniform probability of failure. After each access, the probability of failure is increased by 1 in 10^{-7} to represent wear on the device, which we base on a study by Pinheiro *et al.* [81]. Only full-disk failures are considered since recent work has shown latent sector errors to be surprisingly rare on modern hardware [95]. We do not model catastrophic failure where a restore from tertiary storage is necessary.

We express disk rebuild speed with a single parameter, r, that encompasses the disk bandwidth, network overhead, and CPU load of a disk to form the number of seconds it takes to restore a gigabyte of data. Figure 6.11 shows, unsurprisingly, that larger values of r increase the time lost for failure. We choose $r = 30$ s/GB (≈ 34 MB/s) as our default value for reconstructing data based on the 50 MB/s read speed of an off-the-shelf 7200 RPM disk [63]. This is a low estimate since CPU saturation and not read speed is the typical bottleneck for disk rebuild [63]. We also test on r values as low as 10 s/GB ($\approx$102 MB/s) to demonstrate that PERSES can improve layout even on advanced hardware. We use the **wash** and **water** traces (described in Chapter 3) to demonstrate PERSES. We chose these traces because they had both accesses and semantic information, so we could for the first time provide a direct comparison between statistically and categorically defined groups.

We define *project time lost* as the total delay encountered by all project groups as a result of the access requests that are delayed while disks are rebuilding. The total project time lost can be higher than the total time that the system is in degraded mode since a failure, especially on randomly allocated disks, can affect multiple projects. Once the data is laid out on disk, the trace is played in the simulator to calculate the total

project time lost as a result of failure events. While failures are random, they are set to a consistent random seed between each pair of grouped and random runs to make the runs comparable. In experiments with fewer disks, our results had very high variance regardless of the number of trials because well timed failures can cascade. All random layout runs were run at least 50 and typically over 100 times. One benefit of the PERSES layout in real systems may be the relative predictability of time lost compared to random allocation.

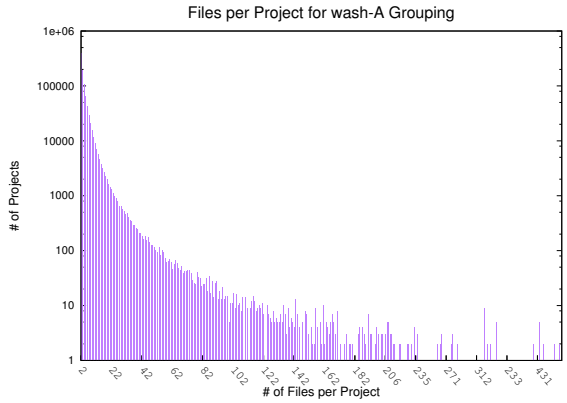
We do not include scrubbing in our simulation because scrubbing, which touches every element it can regardless of semantic qualities, does not satisfy our group association assumption [98]. Members of a group can be scrubbed even if a fraction of the group is offline. The only effect scrubbing is likely to have on project time lost is to slightly increase the failure rate caused by the total number of spin-ups of the drive.

6.2.1 Simulation Results

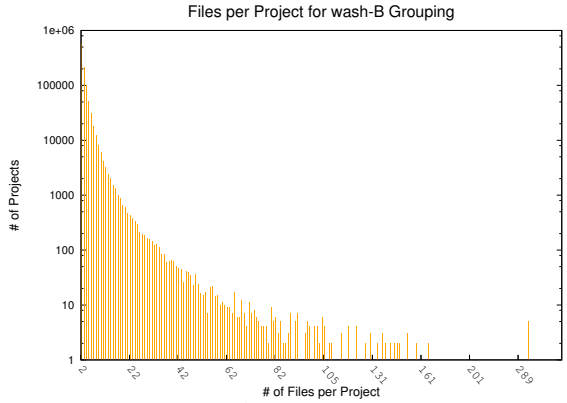
We ran our PERSES simulator on categorically grouped **water** and **wash** data as well as the two statistical groupings of **wash** data. We saw the most improvement from the statistical groupings. To help understand the impact of PERSES, we also averaged the amount of time PERSES gains versus the layout it is being compared against when it has more than two disks over the course of the trace; we call this number *hours gained*.

6.2.1.1 Categorical Grouping

We used principal components analysis to select the most predictive and automatically derivable feature for **water**, which was "Site." The single feature for **wash** was "Record Type." This is consistent with previous work on this data [2, 121]. Figure 6.3 shows the distribution of categorical group sizes for **water** and **wash** data. We see that both graphs exhibit a logarithmic decrease in number of groups as group size increases. Figure 6.4 shows that all of the features used to separate the **water** data set into projects provide very similar project time lost values over the course of the trace. This is surprising because project time lost is determined across all projects. So, over the course of



(a) `wash-A`



(b) `wash-B`

Figure 6.2: Project size distributions for statistical groupings. Both `wash-A` and `wash-B` statistical groupings have enough very small groups to greatly skew the group size distribution towards larger groups. Groups sizes obey an inverse Pareto distribution.

the trace in this case every project sees roughly equivalent impact for failures. This indicates that though the groups are predictive, there is not enough data to correctly isolate projects with grouping.

Figure 6.5 shows that categorical grouping using "Site" on **water** does not significantly reduce the project time lost compared to random allocation for $r = 30$. Furthermore, we found that increasing r, the number of seconds it takes to rebuild a gigabyte of data, to 100 had negligible effect on the relative project time lost. This is likely because the trace is too small to experience enough failures for layout to matter. "Record Type" on **wash** performs better, saving about 1000 hours, largely due to a higher access rate.

6.2.1.2 Statistical Grouping

We used the NNP algorithm to identify statistical projects in our **wash** trace. Groups are determined based on access times and a unique per record ID field that most record possessed. Those that were missing the ID field were classified as ungrouped records. We could not statistically group the **water** trace because there was not enough information to provide a viable second continuous dimension for the classifier.

NNP parameters that affect the grouping include two scaling factors and weights for means and standard deviations in the partitioning calculation. NNP has four parameters that need to be set to obtain a grouping on a dataset. Lacking any external confirmation of validity, we did a complete crawl through the parameter space and then calculated the average group size of all of the resultant groupings.

Table 6.1: Statistics of the two non-trivial groupings NNP found in **wash**

	Avg. Group Size	Standard Dev.	Max. Group Size
`wash-A`	4.7	8.3	1865
`wash-B`	3.2	4.0	1012

When these groupings were clustered using the parameters and average group size as features, elements fell into one of three clusters. The groupings within each cluster

were almost identical. The first cluster, which resulted from extreme parameter combinations, was the "null" grouping where every element is a separate group. We name the representatives we selected from the two non-trivial grouping clusters `wash-A` and `wash-B`. Table 6.1 shows the main differences between the two groupings. Though the difference in average group sizes seems small, the group size distribution within the grouping (Figure 6.2) shows that the inverse Pareto distribution of group sizes results in many more larger project groups for the `wash-A` grouping.

Figure 6.6 shows how allocating data on disk using the `wash-A` grouping improves project time lost by up to 50% at 4 disks going down to approximately 5% as the number of disks increases. This represents an increase of an order of magnitude for larger disks with r, the number of seconds it takes to rebuild a gigabyte of data, as low as 10. At $r = 30$, there is a clear benefit to laying out disks with the `wash-A` grouping with the percent improvement ranging from 10% for small disks to 80% for larger disks. The `wash-B` grouping, on the other hand, did not consistently outperform random allocation (Figure 6.7). This is because `wash-B` has smaller, noisier groups, which biases the layout algorithm towards placing related data farther apart on average. We discuss this in greater depth in Section 6.3.

6.2.1.3 Extensions

To better understand PERSES, we explored the effects of different control layouts and rebuild speeds along with limiting group size and increasing the I/Os per second (IOPS) in our trace. For cache and rebuild, we wanted to explore how PERSES would behave in different system configurations. The minimum group size experiments came from the observation that larger groups had more to lose and potentially experienced more benefit from localization. We found that restricting group size did improve the performance of PERSES. Finally, we realize that our traces have relatively low IOPS, and we wanted to address concerns about the efficacy of PERSES in a more active environment. To do this, we speed up our trace by a factor of ten and show that PERSES improves project time lost even on the compressed trace.

Temporal Layout

We compare our project based layout against a random layout since we have no information as to the actual arrangement of the data behind our traces. To give us another bound on PERSES performance, we also compared it against a "temporal" layout where files are placed on disk by an oracle in the order the files will later be read in the trace. Figure 6.8 shows that even compared to this ideal layout, PERSES at worst matches the project time lost for all but the largest disks, with an average of 1300 hours of project time saved over the three years of the trace.

Caching

Our simulator has a read cache to catch popular reads before they go to disk. We found that the size of this cache, within reason, had very little impact on the project time lost. Figure 6.9 shows that not having any cache at all is competitive with having a cache of 10,000 files for **wash**, which represents 97 GB of cache. This trend continues until the cache becomes so large that every repeat request is served out of cache.

Minimum Group Size

We hypothesized that `wash-A` outperformed `wash-B` because `wash-A` has a higher average group size, and proceeded to test the effect of raising the minimum group size used in PERSES layout.

Figure 6.10 shows both `wash-A` and `wash-B` groupings with all project groups of fewer than either 50 or 100 members removed. The top pair of lines on each graph correspond to a rebuild rate of $r = 30$ and the bottom pair correspond to $r = 10$. Surprisingly, if we restrict the size of groups to only model larger projects, we see significant improvement even with $r = 10$, which corresponds to a rebuild rate of 102 MB/s, with the `wash-A` grouping. Equally surprising, the `wash-B` grouping suffers a significant decline in performance.

This leads to two insights. First, `wash-A` outperforms `wash-B` because `wash-A` has more large groups: with restricted group size, `wash-B` does not have enough information to outperform random. Secondly, the performance of `wash-A` even at $r = 10$ indicates that with a grouping which biases towards larger groups, PERSES is valuable even on

high-end hardware with very fast rebuild. Our results indicate that PERSES has higher impact on larger groups since without group-based allocation there is a greater chance the project will be spread across many disks.

Rebuild Speed

Another question we had was how rebuild speed affected group time lost. Figure 6.11 shows the improvement in project time lost for different rebuild factors. Though low r values do worse due to the overhead of disk arrangement, once values start approaching realistic rebuild rates, relative project time lost decreases as r rises.

High IOPS

Finally, we had some concerns about how PERSES would behave under high IOPS. To test this, we compressed our traces by a factor of 10, such that for example accesses that are 100 s apart in the trace are 10 s apart in the compressed trace. Figure 6.12 shows that while PERSES does not save as much project time lost under compression, it still handily outperforms random allocation once disks are reasonably sized.

6.2.1.4 Re-Replication

We designed PERSES to only store information in the most likely group. In a real implementation, we need to simultaneously optimize for both availability and bandwidth. To improve bandwidth across the system, one could store several groups on disk for every element. Figure 6.13 shows the number of groups elements appear in across all groups, including groups with low likelihood and groups that are subsets of other groups. The average number of groups a grouped element is a member of is about 2.48. Having three replicas of data on a system, in groups or otherwise, significantly improves the reliability of the system as well as distributing the data to avoid network level points of throttling.

6.3 Discussion

Our results show that for most cases, PERSES significantly reduces the time lost across projects in our two workloads.

The most surprising result of our simulations is that statistical grouping significantly outperformed categorical for both workloads. This is because, given the relatively short rebuild period, PERSES favors groupings with high co-access probability. Categorical groupings do not take into account the actual usage of group members, instead only considering semantic similarity. On the other hand, high probability of co-access is one of the defining characteristics of the statistical groups produced by NNP. Over a longer trace, we expect that the categorical groups will perform better, but not overcome statistical groupings as long as there are few sudden and drastic shifts in usage.

The statistical grouping results help illustrate why PERSES works. `wash-A` strictly outperforms `wash-B` because `wash-B` is strongly biased towards small group sizes. In the `wash-B` grouping, the majority of files are members of small projects. These projects, clustered together on relatively few disks, each contribute equally to project time lost on drive failure, inflating the time lost number relative to the `wash-A` grouping. A layout algorithm optimized to distribute small groups across disks could potentially alleviate this issue.

Simply restricting the group size is insufficient because if the grouping biases towards small groups, removing those groups leaves the majority of files without a project association. This is why `wash-B` performs only slightly better than random in the case where the minimum group size is 100 files. `wash-A` performs especially well in the restricted group size case because it biases towards larger groupings, so removing smaller groups removes weak groups more than it adds information.

The bottleneck in rebuilding data is the speed a disk can be read. The rebuild rate for most of our experiments, $r = 30$, corresponds to the 50 MB/s high end disks available today. PERSES will perform even better on slower disks, since each disk failure leads to higher project time lost per project, amplifying the effects of locality. From another perspective, the probability that the failure will be observed by an access increases with

downtime.

6.3.1 Redundancy and Load Balancing

One concern with grouping all of the data a project needs on a small set of physical devices is a slightly higher probability of losing all of the data for a project in a data loss event. There are two ways to mitigate this worry of "putting all ones eggs in the same basket." Though we do not assume a particular reliability method, PERSES is designed for a system with parity groups. These groups can be arranged on a segment level instead of a disk level, distributing the risk. Additionally, our grouping method replicates data that is in multiple groups. This, by definition, replicates data that is useful to multiple projects, making exactly that data that has the greatest net value the most replicated and thus most difficult to lose. Finally, any tertiary storage, such as backups, should be arranged in a project-agnostic layout to provide additional resilience against data loss.

Another issue with grouping all of the data for a project on a small number of devices is that repeated, frequent accesses to the same disk could impact performance while the project is in use. One clear way to address this issue is to place entire groups into a group-aware LRU cache when a member of the group is accessed. All successive accesses to a group are made from memory. In this case, we are not trying to maximize accesses to the cache as much as minimize repeated accesses to a device, so a simple cache policy suffices. Another major concern is the performance impact of rearranging groups on disk when groups are recalculated. Since the grouping loses predictive power slowly over time, groups can be rearranged lazily as disks are used or during low request periods.

6.3.2 Interaction with Existing Technologies

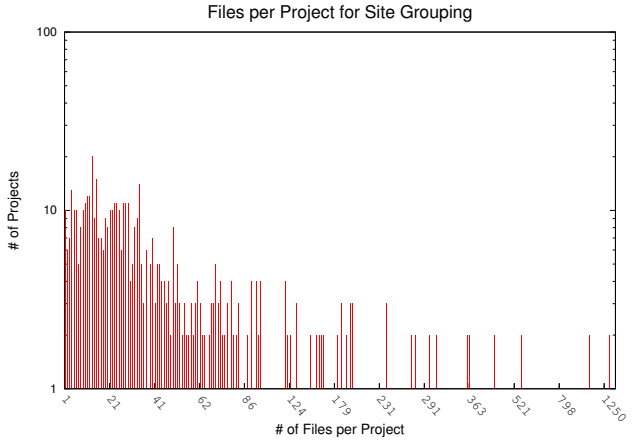
Many strategies exist to improve storage performance during rebuild. PERSES addresses layout, and so will amplify the benefit of current pre-fetching techniques by reducing disk overhead. We cannot outperform strategies that specialize the layout us-

ing external knowledge of the expected workload characteristics; our techniques are designed to be easily adaptable to different, unknown workload types. As more services move to massive multi-project storage systems such as S3, optimization strategies that do not rely on domain knowledge will be essential. In these systems, if user requests are slow enough to time out, the provider will break the SLA, so PERSES applied to the user data has the potential to save cloud providers money and reputation by isolating degradation to few users.

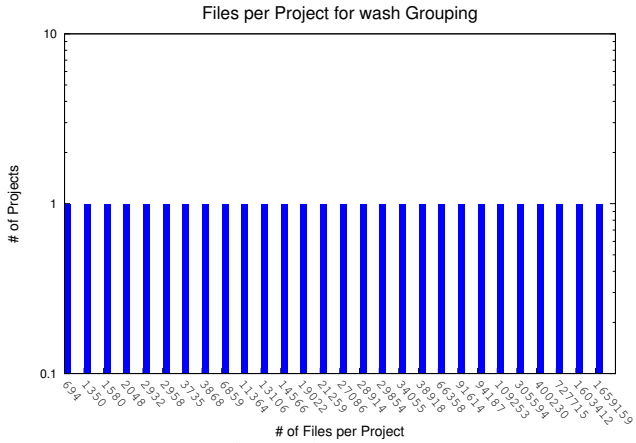
Rapid rebuild is touted as making RAID reconstruction irrelevant, but the latent sector errors it fixes only happen in rapid succession, meaning that it will always be playing catch-up [96]. Secondly, we see improvement with PERSES even at $r = 10$, which corresponds to reconstructing data from a failed drive at over 100 MB/s, faster than most drives can even be read. Finally, we also experimented with modifying the read cache size and found that it did not significantly alter our results. This indicates that layout models based on the popularity of data for placement would have performed poorly on our traces.

Based on the results from increasing the IOPS of our data, we can extrapolate our results to workloads with more activity such as enterprise or HPC. As IOPS increases, PERSES still saves several hundred hours in project time over three years. In a real high IOPS trace, this number would likely be higher because instead of projects being accessed faster, more projects are being accessed. Thus, the time lost should be closer to the normal case than the high IOPS case we simulate here. Finding data sets with labels for categorical grouping is very challenging. While our current results are compelling, we are actively seeking other workload types to test.

Performance under PERSES is much more predictable than under arbitrary layouts, which could allow system administrators to set better SLAs and predict patterns in their workflows. PERSES is general purpose and requires no administrative overhead to find groups or re-arrange data, and it can combine with existing rapid reconstruction methods to help alleviate the performance impact of disk rebuild, and may help RAID-like systems scale to match the demands of the cloud.



(a) Water Dataset



(b) Wash Dataset

Figure 6.3: Grouping size distribution for Categorical Groups. Both datasets show a similar distribution of group sizes, with the **wash** dataset having much larger groups overall. Note that the y-axis is on a log scale and the x-axis corresponds to individual groups.

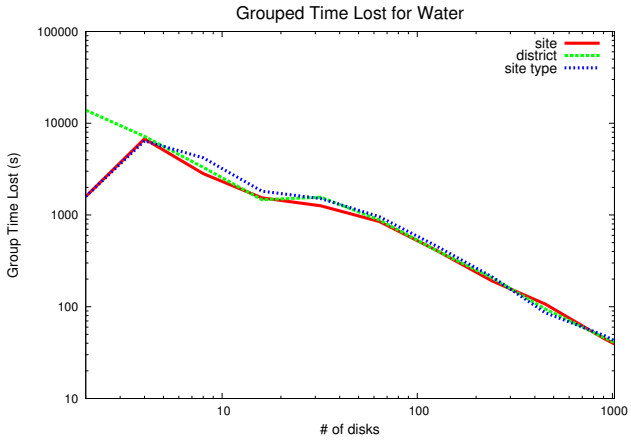
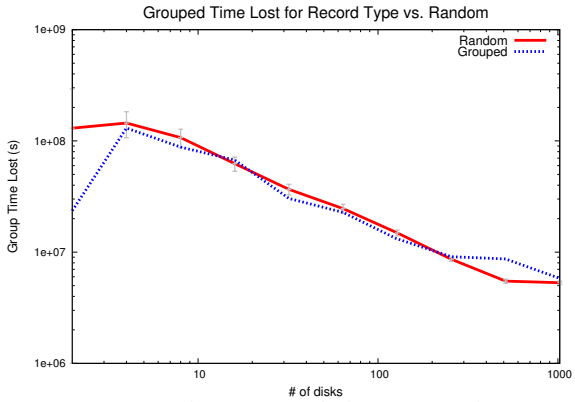
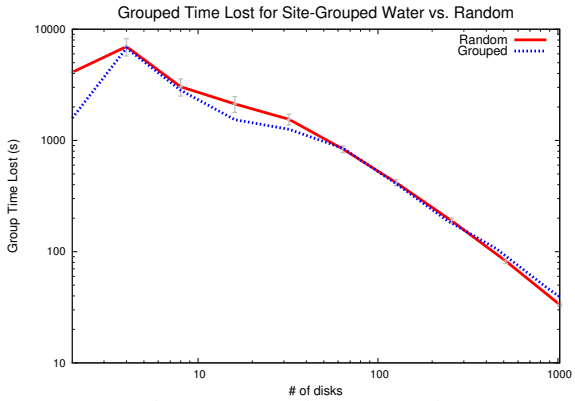


Figure 6.4: All of the categorical labels in the water trace provide similar grouped time lost trends.

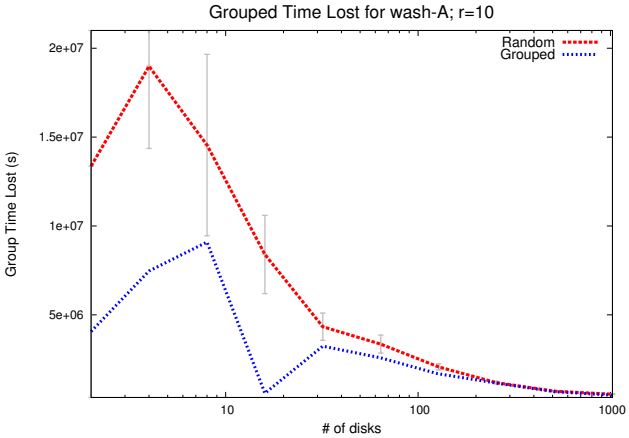


(a) Washington: 1060 hours gained

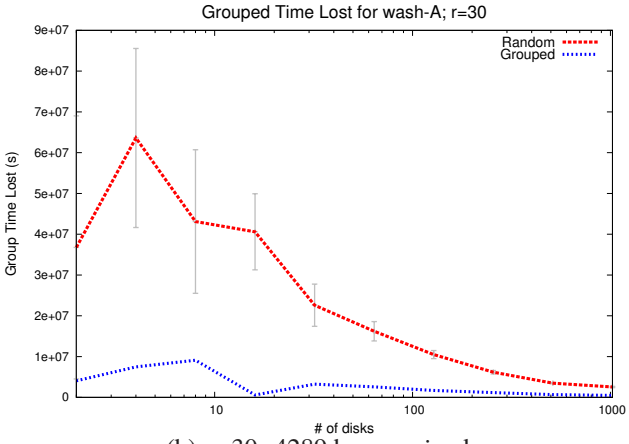


(b) Water: .09 hours gained

Figure 6.5: These graphs show that neither categorical grouping significantly improves on random arrangement at $r = 30$.

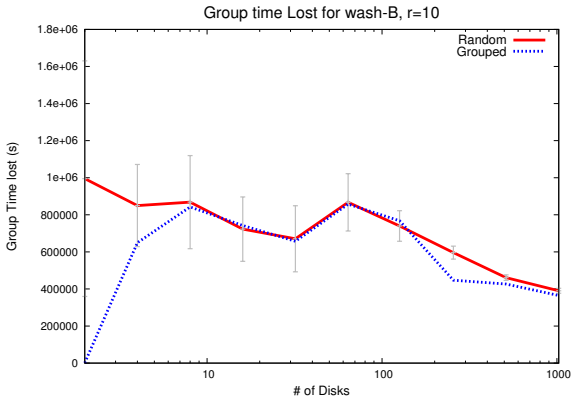


(a) r=10: 837 hours gained

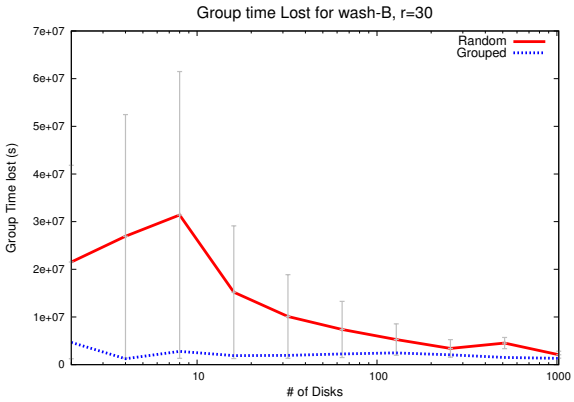


(b) r=30: 4289 hours gained

Figure 6.6: PERSES significantly reduces project time lost with the `wash-A` statistical grouping.



(a) r=10: 12 hours gained



(b) r=30: 2740 hours gained

Figure 6.7: PERSES shows less improvement with the `wash-B` grouping.

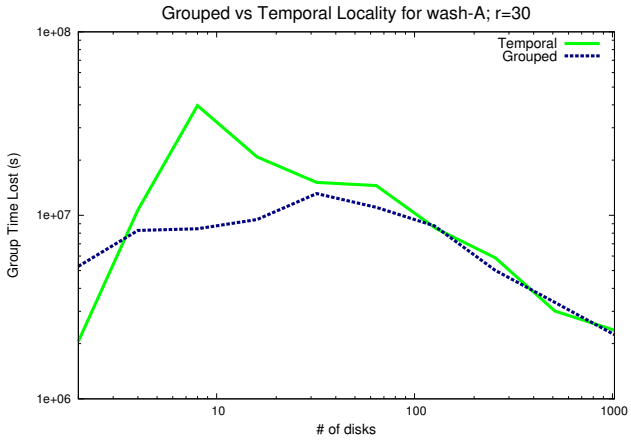


Figure 6.8: PERSES performs as well as or better then data allocated with a temporal oracle. 1321 hours gained.

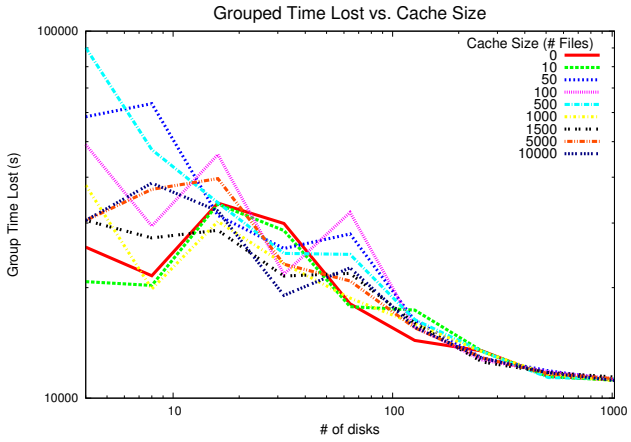
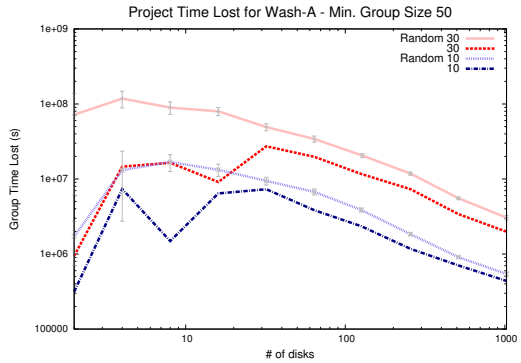
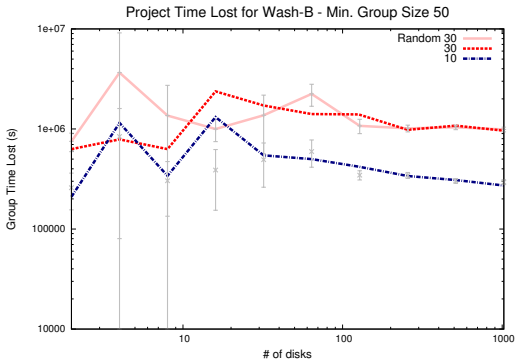


Figure 6.9: Cache size has negligible effect on project time lost. This graph was made with $r = 30$ and grouping `wash-B`, but other parameter combinations produced similar results.

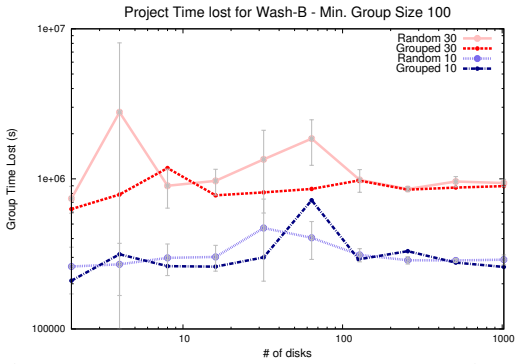


(a) Min. Group Size 50, `wash-A`: 9252 hours gained for $r = 30$



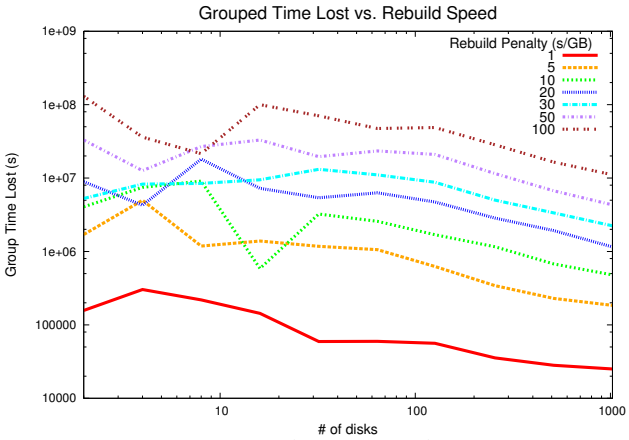
(b) Min. Group Size 50, `wash-B`: 74 hours gained for $r = 30$

(c) Min. Group Size 100, `wash-A`: 9214 hours gained for $r = 30$

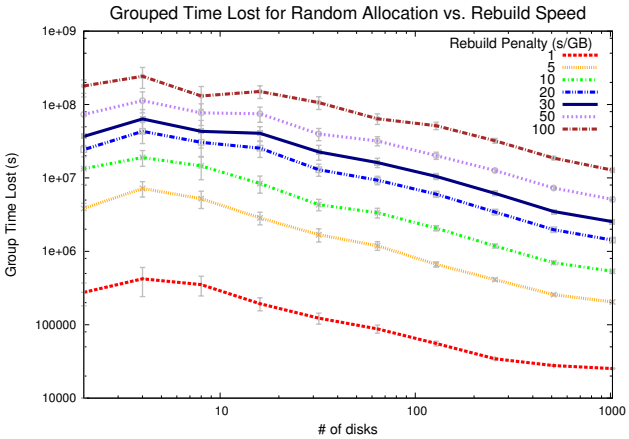


(d) Min. Group Size 100, `wash-B`: 110 hours gained for $r = 30$

Figure 6.10: `wash-A` saves almost three times as many hours with restricted group size, but `wash-B` suffers a significant penalty. Time lost is on a log scale.



(a) Wash-A Grouped



(b) Random

Figure 6.11: With fewer, larger disks, PERSES benefits from high variance accesses at high rebuild rates.

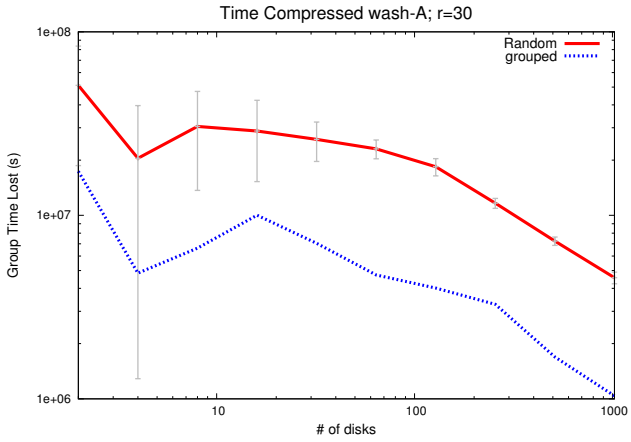


Figure 6.12: Compressing the accesses in `wash-A` by a factor of 10 only slightly hurts the performance of PERSES

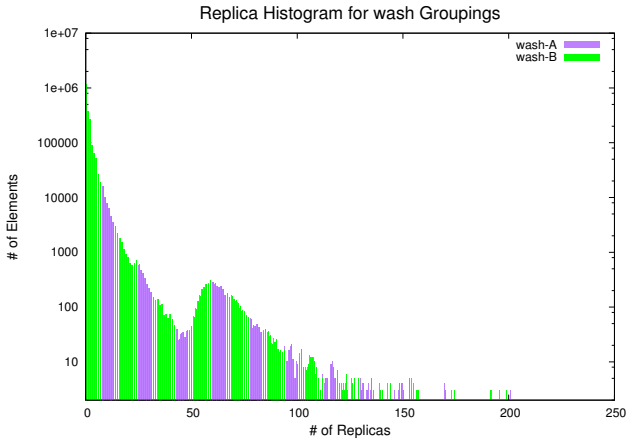


Figure 6.13: Elements overwhelmingly appear in only one group, but can appear in as many as 300. Note that this includes groups that have low likelihood.

Chapter 7

Grouping for Data Reduction

It's the data, stupid.

– Jim Gray

Though the price of storage is falling rapidly, the amount of data we are generating is outpacing the storage space that many organizations are willing or able to afford and maintain [31]. Our global society is already creating more data than we are creating storage, and these numbers are only expected to diverge further as new scientific apparati come online and more businesses leave behind legacy filing cabinets. Figure 7.1 shows that the projected rise in data production compared to storage production. Some analysts believe that over half of the information we currently produce does not have a permanent home [58].

To address a growing need for primary space savings, researchers such as Constantinescu [18], Jones [49], the iDedup team [104], and the DBLK team [113] have proposed deduplicating primary storage in addition to backups and archival systems. At a high level, deduplication compares segments of data against an index to determine if the segment in question is already stored on a system. Unlike backups, primary storage is constantly accessed, so the deduplication system must check segments against the index shortly after the write request. This is known as in-line deduplication.

In-line deduplication is rarely used on primary storage because checking for duplicate blocks has a significant performance impact. Many groups have tried to reduce

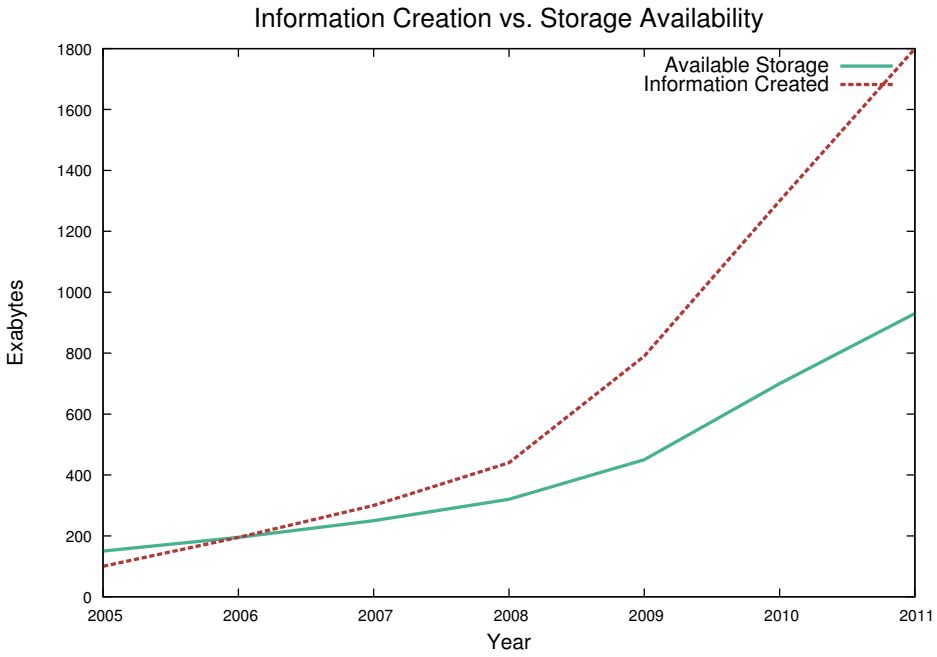


Figure 7.1: Worldwide, data is being created much faster than storage capacity. Note the zettabyte gap between information created and new storage created in 2011. Data source: IDC [36]

this impact by moving parts of the index to memory, but the cost of storing the index in memory scales with the amount of unique data in the storage system and quickly becomes economically infeasible. In a system where some of the index must remain on disk, performance is impacted by paging to and from disk: this is known as the *disk bottleneck* [11].

To illustrate the cost of memory, suppose we want to deduplicate incoming writes to a system with 100 TB of unique data and a data segment size of 4 KB, resulting in 2.7 billion segments. At an average of 32 bytes per segment hash, this would result in a massive 800 GB of memory to store the deduplication index. At $100 per terabyte of disk and $10 per gigabyte of DRAM, the $10,000 disk array would require $8,000 of dedicated memory to store the entire deduplication index. This means that if the deduplication rate for the workload is under 45%, which is more than many primary deduplication systems achieve [104], we are actually paying extra to store less data with primary deduplication. To make matters worse, there is a strong argument for using 1 KB instead of 4 KB segments, which increases the break-even point for deduplication from 45% to an effectively unattainable 76% [18].

To make deduplication on primary storage economically feasible we introduce HANDS: a scalable, in-line, chunk-based deduplication framework for primary storage. HANDS generates correlated groups of segments, or working sets, based on usage patterns and places the corresponding segment hashes, or fingerprints, adjacently in the index cache so the entire group can be accessed atomically when an element is accessed. The method we propose for group identification relies only on historical segment I/O access data, which is easy to collect and interpret across diverse storage environments including HPC and enterprise. Though we propose a domain agnostic method for determining groups, our technique does not rely on any particular method of generating groupings.

We tested HANDS on trace data from a large, multi-user enterprise storage system as well as a university research server and found that loading groups of data into the index cache significantly reduces the number of accesses to the on-disk index cache while realizing most of the data reduction potential.

Much of the data in large systems, generated data in particular, has a high duplication rate [18]. Some workloads, such as virtual machines, have obvious massive duplication. Other workloads, such as scientific computing, have little file level duplication since results are unique but have a high degree of chunk level duplication as there may be very little difference in the results files between experiments. To take advantage of this duplication to control exploding data footprints, researchers such as Chambliss [18] and the DBLK team [113] have began to use de-duplication on primary storage in addition to backups and archival systems. We tested HANDS on trace data from a large, multi-user enterprise storage system as well as a university research server and found that loading groups of data into the index cache significantly reduces the number of accesses to the on-disk index cache while realizing most of the data reduction potential.

Our work presents two major contributions. First, we demonstrate a dynamic, scalable method to select a portion of the deduplication index to store in memory, reducing the memory cost for primary deduplication by up to 99%. Second, we provide an experimental evaluation of deduplication with different methods of in-memory index management and memory footprint size, demonstrating the generality and adaptability of our technique. We found that we can reduce the percentage of the index cache that is stored in memory to 1% while still achieving 90% of the optimal space savings for stored data. We also found that only 10% of the total data blocks read by a trace need to be pulled into the memory cache at all to achieve reasonable results.

We note that with HANDS, the reliability of data is now inversely proportional to the popularity of the most common chunk. There have been many methods proposed to replicate data post-de-duplication based on popularity, and we recommend implementation of such a technique in any primary de-duplicated system. Another issue is that backups or snapshots of the data will be expanded to full size without another de-duplication system in place in the backup pipeline. Even with a data reduction system, depending on the order of data transfer to secondary storage the overall data footprint could get very large. This is significant work to be done with making our group de-duplication system compatible with secondary storage.

7.1 In-line Deduplication

The goal of any deduplication method is to identify redundancy in storage and eliminate it. In Section 2.3.3, we discuss work that has been done in increasing the efficiency of deduplication of data streams, such as for backups. This work generally does not apply to our problem because backup workloads have a known stream ordering and different performance constraints. This deduplication is not on the user pathway, so though it needs to be efficient it is less performance sensitive than primary storage deduplication. Primary deduplication is "in-line" because writes are deduplicated as they come in versus stored and deduplicated in batches.

In-line data deduplication generally consists of three steps per incoming data segment:

1. Identify whether the segment is a duplicate
2. If the segment is a duplicate, create an index pointer for that write
3. If the segment is new, store the data and add its fingerprint to the deduplication index

The critical step that our work addresses is the need to access and potentially update the index without increasing the perceived I/O write time. Manipulating memory is relatively imperceptible, so we focus our efforts on moving as many of the accesses as possible from disk into memory.

Much of the data in large systems, generated data in particular, has a high duplication rate [18, 72]. Some workloads, such as virtual machines, have obvious massive duplication, and these images do not get cleaned up the way localized storage does [47]. Other workloads, such as scientific computing, have little file level duplication since results are unique, but they have a high level of chunk level deduplication from the results being similar or accretive.

Scalable primary storage has become more of a concern as the concept of "stale data" evolves. While organizations used to be able to identify older data to store on

tertiary storage, many modern datasets exhibit an unpredictable long-tail access pattern, creating an archival-type workload where the write-once, read-maybe assumption no longer holds [2].

While some groups have presented solutions for backup and archival workloads, primary deduplication at the peta and exabyte level remains relatively unstudied [11, 32]. The main argument against deduplication on primary storage is the performance impact on the system caused by hash calculation and additional I/O requests, particulary the disk-bottleneck problem [18].

7.2 Working Set Identification

A good group identification algorithm for HANDS produces groups that are small, making them less likely to churn the cache and more likely to have high predictive value. The algorithm must also avoid overfitting to the training data. Finally, any technique used for grouping has to have low overhead and produce a grouping lookup table with a small memory footprint.

For this project, we modified the techniques from Chapter 4 to accommodate the very large enterprise storage traces using a limited memory environment. First, we break up traces into neighborhoods, the size of which is determined by the memory available to the grouping process. Using the $m \times m$ distance matrices and neighborhood partitioning as discussed in section 2.1, we calculate one matrix and related grouping per neighborhood. Once all of these neighborhood groupings are calculated we need to combine them. We do this by searching through the lists of groups and selecting only those groups that appear in all the lists or those groups and groups whose elements never appear in other groups. For example, for two group lists $[(x_1, x_4, x_7)(x_1, x_5)(x_8, x_7)]$ and $[(x_1, x_3, x_7)(x_1, x_5)(x_2, x_9)]$, the resulting grouping would be $[(x_1, x_5)(x_2, x_9)]$. We choose this aggregation technique because it has a natural bias against larger groups; this in turn limits the amount of churn in our cache. To avoid an additional lookup during runtime, we must store our groups as sets of fingerprints instead of sets of LBAs.

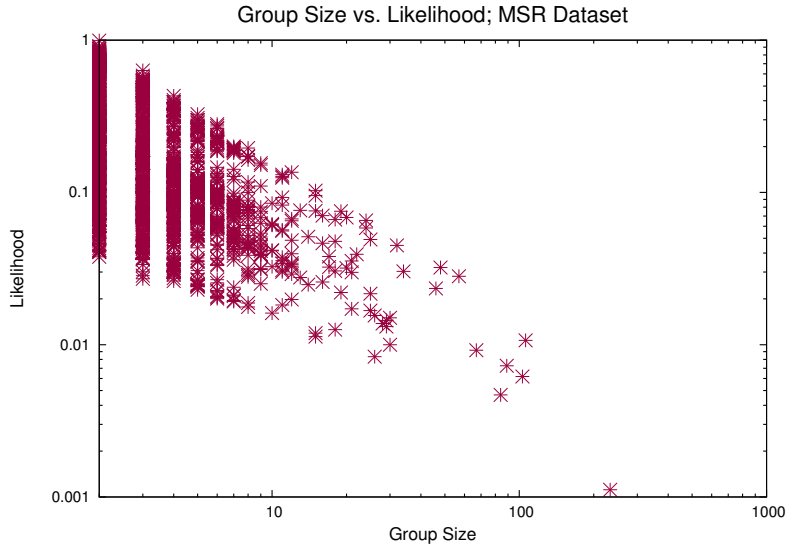
This group calculation happens in the background during periods of low activity. As accesses come in, we need to update groups to reflect a changing reality. We do this by storing a likelihood value for every group. This numerical value starts as the median intergroup distance value. If a requested fingerprint appears in multiple groups, only the group with the highest likelihood is returned. This serves to further augment the bias towards small groupings, which we have found to have a higher average likelihood. We re-calculate the groupings and the associated likelihood values once per day.

In a real system as well as in our simulation, it is essential that the working sets an element is a member of can be quickly referenced.

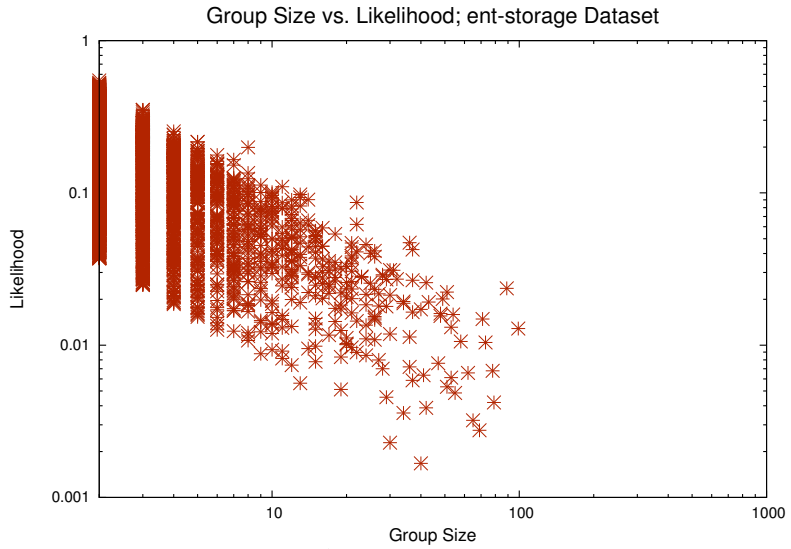
Most workloads on most systems we observed showed a group distribution that was heavily biased towards small groups. Additionally, when calculating likelihood values over groups, small sets had higher average likelihood (Figure 7.2). Given this, we arrange our group data structure as a tiered hashtable. The upper tier maps group sizes to a group of groups while the second tier maps fingerprints to the appropriate group.

7.3 Design

HANDS is a framework for content addressed in-line deduplication that incorporate groups to manage the memory footprint of the fingerprint cache. These algorithms can be interchanged modularly so, for example, another grouping technique could easily be substituted to tune HANDS for a particular environment. Our high-level framework consists of three elements: the *fingerprint index* mapping fingerprints to chunks, the *index cache*, which is a subset of the fingerprint index that is kept in memory, and the *working set table* that maps fingerprints to groups of fingerprints. The index cache is managed using LRU or LFU caching in this work, but the framework does not depend on the cache replacement policy used.



(a) **fiu**



(b) **ent-storage**

Figure 7.2: In both NNP groupings for HANDS, the most likely groupings tended to be smaller at time of group calculation. These graphs represent a subsample of 10% of groups per grouping, randomly selected. Likelihood is normalized within each grouping.

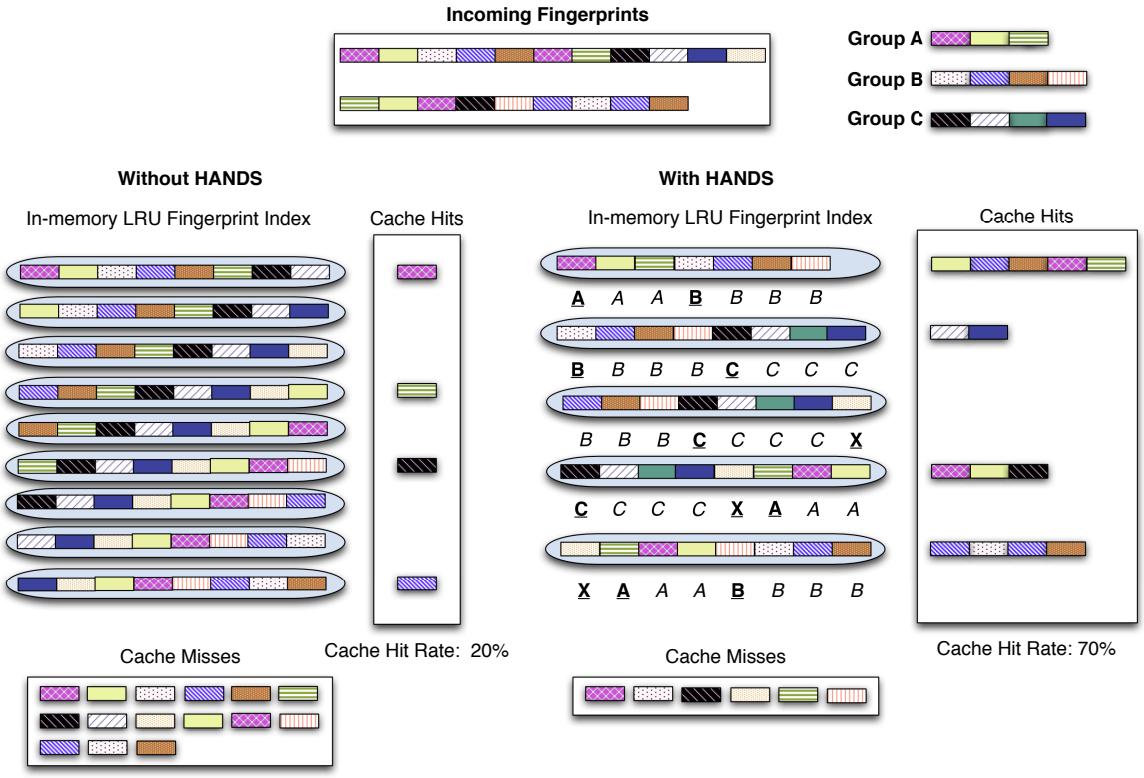


Figure 7.3: Illustration of HANDS on a toy example. The patterned rectangles correspond to fingerprints. We see that adding three groups, or working sets, improves the cache hit rate significantly by pre-fetching fingerprints into memory. In the diagram, the letters underneath the fingerprints correspond to group membership where the bolded letter is a group member that required a disk seek (a cache miss) and the italicized letters are group members that were pulled in with a bolded member.

7.3.1 Initialization

The first step towards deduplication is creating the working set table. Our method for compiling this table is covered in Section 2.1, but any grouping mechanism that results in groups that are small and predictive can be substituted. The next step is to allocate the fingerprint index and the index cache. The index cache is fixed size, allocated in memory, and starts out empty. We explored bootstrapping the index cache with the most frequently accessed or highest likelihood groups from training data, but found

that neither improved our results. The groups are then written serially into the on-disk fingerprint index such that entire working sets can be retrieved without seeking. Since fingerprints can be members of several groups, this could lead to duplication within the on-disk fingerprint index. We accept this because the small amount of extra index cache required is inexpensive compared to the memory savings. Since a fingerprint could potentially be in several groups, this could result in a small data overhead which is acceptable because disk is relatively inexpensive.

7.3.2 Deduplication

Once the indices are established, we can begin de-duplicating incoming I/O requests. We expected most of our benefit to come from having the fingerprints for incoming write requests in cache instead of the main fingerprint index on disk. While there is some conceivable benefit to also calculating fingerprints for chunks read, that benefit is highly dependent on the underlying data retrieval mechanisms, and thus we limit the scope of this paper to write requests. We also assume that all the accesses we get are post disk cache.

Figure 7.4 outlines the interactions of the components of our deduplication system. Our first step on receiving a write request is to calculate a hash value for the write. Any fast, low-collision hash method works equally well; we recommend SHA-1, which for an exabyte scale system has under a $1 \times 10^{-16}\%$ probability of hash collision in non-adversarial situations while still having a fast runtime [47, 22].

7.3.2.1 Duplicate Data

After the hash is computed, we compare the hash to the fingerprints in the index cache. If the hash is found in the index cache, the request is identified as a duplicate and the index cache is updated according to the caching algorithm in use. We refer to this as a *cache hit*.

In the case of a *cache miss*, the request is sent to the on-disk fingerprint cache, which is arranged in tiers as shown in Figure 7.5. There is a Bloom filter across the entire index

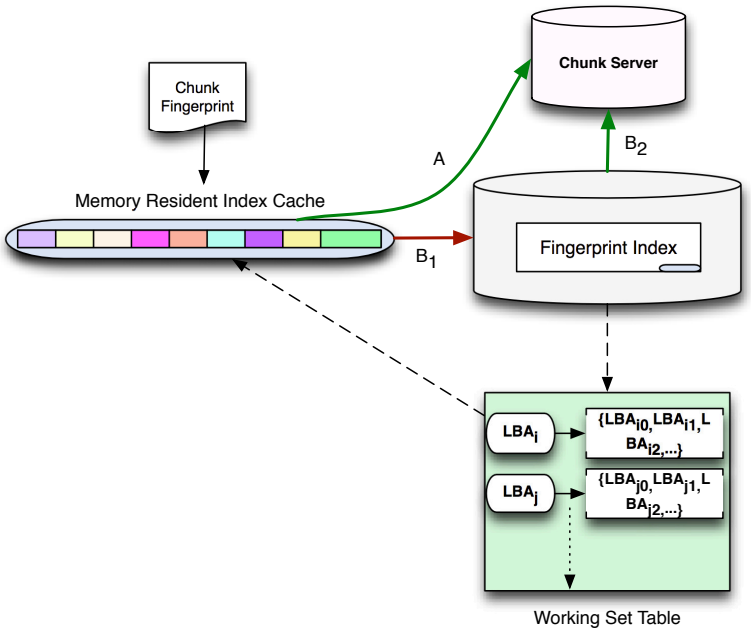


Figure 7.4: Deduplication framework. A duplicate chunk is either in the index cache (path A) or must be recovered from disk (path B). When a fingerprint enters path B, the group for that fingerprint is pulled into the index cache from the fingerprint index, which is laid out in group order.

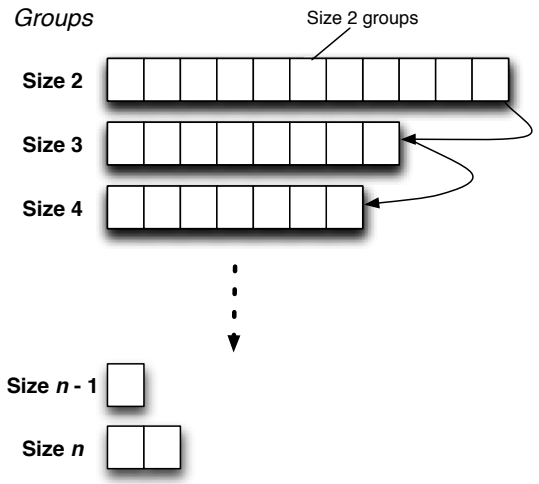


Figure 7.5: The fingerprint index is tiered so groups of size $n-1$ are searched before groups of size n

to quickly detect new data. Each tier represents groups of a given size, and they are searched from smallest to largest until a group is found. The bias towards small groups here is intentional and designed to limit cache churn.

If the fingerprint is found in the fingerprint index, in addition to serving the content the system also queries the working set table to see if the fingerprint has any known groups. We accept the first working set match for a hash where the likelihood value is above the mean likelihood minus one standard deviation. Smaller groups are more likely to have high likelihood, so our tiered group index (Figure 7.5) reduces the search time in very large on-disk indices. The group returned by the working set cache is then copied into the index cache; overflow cache contents are removed based on the caching algorithm replacement policy. While this results in some CPU overhead, we accept this because it is negligible compared to the penalty for the I/O.

7.3.2.2 New Data

If a fingerprint is not in the fingerprint index, the data is written and the hash is stored to the fingerprint index. New writes are not members of any groups, so they can safely be placed in a temporary area without being accidently pulled into memory as a group

member. When working sets are next computed, the fingerprints will be added to groups as appropriate. Fingerprints on disk are co-located based on group membership. Since fingerprints can be members of several groups, this could lead to duplication within the on-disk fingerprint index. We accept this because the small amount of extra index cache required is inexpensive compared to the memory savings.

7.3.3 Design Considerations

Figure 7.3 shows how HANDS noticeably improves the cache hit rate. The rectangles represent fingerprints, uniquely identified by color and pattern. There are three groups, identified by the letters underneath the fingerprints. An LRU cache without HANDS (left) catches most quickly repeated fingerprints and must go to disk for everything else. With HANDS (right), the cache predicts future fingerprint accesses and thus achieves considerably better cache hit rate. There is a concern that large groups could fill the cache quickly, causing a high amount of cache churn that would push out relevant data. With the group calculation methodology we propose, however, the likelihood of a group tends to decrease with every additional member. This inherent bias towards smaller groups led to less cache churn in our experimental results.

Fingerprints and blocks do not share a fixed mapping. In fact, for one of our workloads we found that over 78% of the time, consecutive accesses of the same block had different fingerprints. Thus, we group using block address but need data with actual fingerprints in order to estimate index cache performance. We found experimentally that groups tend to stay the same over time even as the fingerprints associated with blocks change [123]. Therefore, we hypothesize that our block addresses are generally "unique under mutation," meaning that the usage of the data stays similar even as the actual data is modified.

7.4 Experiments and Results

To test the HANDS design, we simulated a deduplication environment where a portion of the fingerprint index is stored in memory both with and without the addition of groups. We pass real traces with fingerprint hashes through the simulator to determine the efficacy of the group based cache.

We measured the effect of HANDS grouping using three cache replacement algorithms: LRU, naïve LFU (LFU), and working set aware LFU (LFU-ws). Our implementation of LRU is straightforward; the oldest elements are dropped successively until there is enough room for the new element. Elements of the same age (*e.g.* members of a group that were pulled in together) are dropped in the order in which they appear on disk, which is preserved in the cache. Naïve LFU is an approximation of LFU that drops the least frequently used elements in the cache and is unaware of out-of-band relationships between data once the data is in the index cache. As a result, after a cache hit only the accessed element has its frequency value updated. When the LFU-ws algorithm records a hit to the index cache, the access frequency for every member of the group associated with the accessed fingerprint and currently in cache is updated by .5, while the frequency of the accessed fingerprint is updated by 1 as usual. This biases the algorithm towards keeping groups together in cache and quickly throwing out "singleton" accesses that have no group and are not rapidly accessed. We did not further tune this parameter because LFU in general was unpromising.

7.4.1 Data

We tested our design using data sets from two real systems, **fiu** and **ent-storage**, which we describe in Chapter 3. As a reminder, the **ent-storage** dataset comes from a GPFS server that held researcher data. This storage server has 120 TB of disk along with a 60 GB cache. `fiu` is from Florida International University and traces local researchers' storage [52]. For both of these traces, we used timestamp and logical block address (LBA) to create groups. For our purposes, the most important difference between these

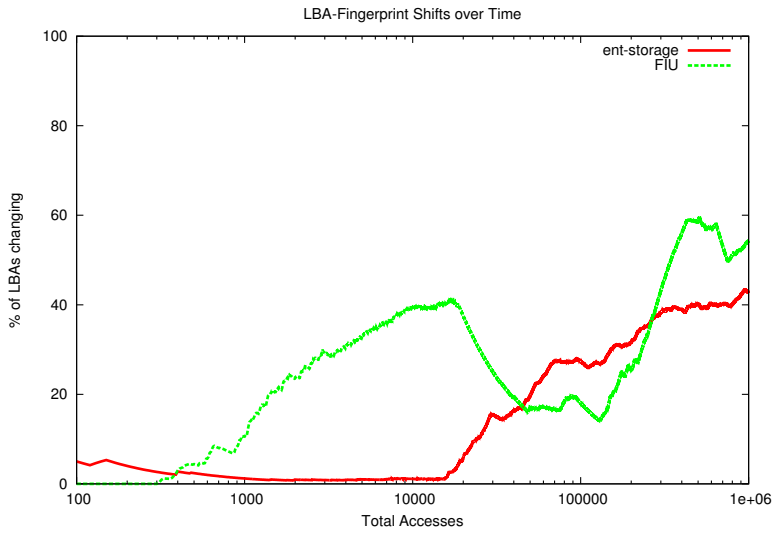


Figure 7.6: Without content data, the meaning of an LBA will shift over time. Removing write accesses significantly lowers the rate of LBA-shifts.

two data sets was average IOPS, meaning that over time it was harder to get predictive groups for `ent-storage` than `fiu`.

One concern with this method is that the groups are made without the content data in mind. For example, `fiu` had *data shifts*, which are consecutive accesses of an LBA were associated with different fingerprints, 78% of the time. In this case, however, there were significantly more write accesses than read accesses, and we believe that the different fingerprints on consecutive LBAs thus represent different content with the same access characteristics (Fig. 7.6). If we take a subset of `fiu` with a higher read ratio (still only 9.5%), the number of data shifts drops down to 58.5%. Though our current results are not positive, there is promising work here, especially with workloads such as VM images that have higher duplication ratios.

7.4.2 Results

We use competitive analysis [12] to compare our algorithm against an ideal cache, and we present graphs as percent of *ideal cache* versus percent of total fingerprint

size.We measure ideal cache as the best a cache could do if it could always recall an element it has seen before, *i.e.* if the entire fingerprint index were in memory. Total fingerprint size is the sum of all of the unique fingerprints over an entire trace. We realize that this may underestimate the data size a real system needs to handle: all of our data is accessed, which is not true in many systems. However, this is the best representation that was available to us and is useful for our work because we are primarily interested in predicting accesses to elements we have seen before.

7.4.2.1 LRU

LRU was the best cache replacement strategy for the index cache with working sets. Figure 7.7(a) shows that the cache hit rate for the `fiu` dataset was almost ideal even with an index cache that was only .01% of the total fingerprint size. In contrast, without groups `fiu` had an unsurprisingly steady increase in index cache hits as the cache size increased. Adding groups to LRU in the `fiu` case worked so well because `fiu` almost entirely represents accesses by real people and so has a very high degree of temporal locality. The `fiu` workload also has far lower IOPS, which helps groups remain stable over time.

In the `ent-storage` dataset (Fig. 7.7(b)), we see a more modest but still clear improvement in cache hit rate for every cache size except about .05% after adding groups. At .05%, we see the beginnings of cache churn: the phenomenon where the cache is too small to hold all of the elements that are being accessed and so is passing elements in and out. We see in Figure 7.9(b) that our `ent-storage` cache with a cache size of .01% begins to churn at about 1.4×10^6 accesses. We can reduce this churn in the future by modifying the LRU to remove entire groups at a time instead of just elements. We also note that the group line never dips below the base LRU line, implying that the cache churn is not severe enough to impact the base LRU performance.

7.4.2.2 LFU

Figure 7.8 shows that for both the `fiu` and `ent-storage` datasets, there is a substantially smaller improvement in cache hit rate when grouping is added to the LFU caching algorithm as compared to LRU. This is surprising at first glance, but it is logical when the effect that the cache replacement policies have on groups are taken into account. The benefit of groups for access prediction comes from a heightened probability of co-access within a group in a given period of time. LFU evicts group members almost as soon as they are pulled in, since they are often not used immediately. Indeed, if they were used immediately, they would be trivially easy to find and much less interesting. We attribute the slight success of LFU on the `fiu` dataset to the presence of a large number of sequential working sets. Sequential accesses were automatically filtered out of the `ent-storage` trace before it was given to us, so LFU underperforms. Note that even without sequential accesses the grouped line never falls below the ungrouped line, indicating that the working sets are not pushing enough other predictive elements out of cache to impact the base, working set free, performance.

We also ran our simulations with LFU-ws, but saw essentially identical results compared to LFU, and thus do not include those in this paper. The results were likely identical because the bias provided by LFU-ws is not enough to offset the huge disadvantage of having groups pushed out of cache quickly. We believe it is worth investigating whether there is a balance, but we reserve that for future work.

7.4.2.3 Random Working Sets

We also implemented a random working set generator to compare HANDS against. Our goal was to identify any unforeseen externalities in pulling large chunks of data into the index cache created in our deduplication system. To best mirror our observed group distributions, we wrote a random generator sampling from a discretized Pareto distribution. This has similar properties to the discrete zeta-distribution while being computationally simpler.

The Pareto distribution is sampled from uniform using the formula:

$$X = \frac{x^m}{U^{1/\alpha}}$$

Here, x^m is a parameter indicating the minimum value of X, which for groups is 1, U represents the uniform distribution between $[0, 1]$, and α is a shaping parameter. We set $\alpha = 3$ to replicate our small-group bias. The resulting continuous value is then rounded to obtain an integer group size. This distribution strongly skews towards small values with few outliers and thus is a good fit for our small groups (Fig. 7.10).

When we ran this, however, we found that the results were identical to the ungrouped results. We attribute this to the large search space of LBAs combined with the small size of groups resulting in an exceptionally low probability of successful access. Indeed, the `ent-storage` case had 0 predictive fingerprints while the `fiu` random run had under 5%. Thus, we can say with some confidence that the benefit of groups is more than just pulling extra data into cache; the pre-computed correlation of data in groups has value.

7.5 Analysis

Throughout this project, our technique performed as well as or better on the `fiu` dataset compared to the `ent-storage` dataset. We learned that, in the `ent-storage` dataset, sequential accesses are not part of the trace we were given because they are pre-fetched by the storage hardware. Since previous work has shown that sequential groups are common and strong groupings, we thought that the lack of sequentiality in the `ent-storage` dataset was to blame for its relatively poor showing in both the LRU and LFU cases. However, in Figure 7.9(b) we see an inverse relationship in the `ent-storage` dataset for the LRU case; there is a strong correlation between groups becoming sequential and groups becoming less predictive. In the parallel figure for the `fiu` dataset (Fig. 7.9(a)), we see no relationship between the sequentiality and the percentage of predictive accesses. Instead, the difference in the two datasets in the LRU case is likely a consequence of the average IOPS of `ent-storage` being high enough to make groups more transient.

We also verified our theory that the LFU `ent-storage` case suffered from cache churn by tracking correct predictions over time based on cache size. In Figure 7.13, we see that the few accesses that have a chance to be predictive are correct for small cache sizes before taking a precipitous drop as the cache size grows. This indicates two things: first, that groups are being evicted early on, leading to an inflated rate of correct predictions, and second, that as the cache grows there are enough legitimate fingerprints being removed from cache that even an improvement in longevity of group members is not enough to salvage the algorithm. LFU is simply a poor choice for a deduplication index cache with groups.

Figure 7.14(a) shows that smaller cache sizes pull in more LBAs early on, but over time the total number of LBAs pulled converges. Even though the average increase in cache hit rate for adding groups to `ent-storage` for LRU was only about 10%, we accomplish this by only pulling in 3% of the total LBAs. For the `fiu` case, we see that at about one million accesses, where the `ent-storage` trace ends, about 15% of the total LBAs are pulled in regardless of cache size. In contrast to `ent-storage`, by the end of the trace almost 80% of the total dataset LBAs had been pulled in. This high percentage of LBAs in cache is almost certainly why the cache performed so well for the `fiu` with LRU case.

As we see in Figure 7.11, for LRU caching the `fiu` dataset pulled in significantly more sequential groups than `ent-storage`. We have seen in our previous work that sequential groups are common and strong groupings, and so we believed sequentiality is the key difference between how amenable these two datasets are to working set deduplication.

One concern with this method is that the groups are made without the content data in mind. As we see in Figure 7.15, the fingerprint-LBA pairing is transient. If this pairing falls away before groups are re-calculated, the quality of predictions could decline. In this case, however, there were significantly more write accesses than read accesses, and so we have good reason to believe that the different fingerprints on consecutive LBAs thus represent different content with the same access characteristics. In a subset of `fiu`

with a slightly higher read ratio of 9.5%, the maximum percent of LBAs changed drop from 78% to 58.5%. Even with our high read ratio, we see that for both the `fiu` and `ent-storage` datasets the correlations between LBAs and fingerprints remain fairly consistent until the system sees about 500,000 accesses. We are interested in acquiring more datasets with fingerprints to determine how this compares to other types of workloads.

Determining when to recalculate the groupings will be essential to future real-time systems. Though we did not recalculate groupings often during the course of our runs because we had relatively little data, we looked for insights to determine when to recalculate our groupings. Figures 7.9(b) and 7.9(a) show that the predictive power of our groupings is strongest early on. A real system could have an automatic alert system to track the level of predictive power and re-calculate groups in the background as needed. Alternatively, working groups could be calculated even more frequently to correspond with the need to go to disk to fetch recent LBA-fingerprint pairs. As we see in Figure 7.15, the LBAs and fingerprints shift at about 500,000 accesses in both traces, though the `fiu` trace stops shifting for a time after. More frequent group calculation is likely to slightly improve cache hit rate numbers as the groups will more closely match the current working environment. However, since groups are based on a somewhat longer term system view, re-calculation should not provide a large bonus in the absence of a major usage shift.

7.5.1 Overhead

While keeping the working set table up to date results in some CPU overhead, we accept this because group calculation will never cause I/O blocking. We also used NNP, an $O(n)$ grouping algorithm that runs in our simulator in under ten minutes. Our implementation was done on a personal desktop using Python for both group calculation and cache simulation. Figure 7.16 shows the overhead for LRU for `ent-storage` and LFU for `fiu`. These graphs are representative of all of the experiments we did, and show that while the grouped version takes about twice as long, the grouped and ungrouped lines closely track each other. This indicates that overhead is mostly fixed and can be

predicted. Though we used the PyPy high-speed Python implementation [87], our code is designed for prototyping. As a result, the runtime numbers we have should only be considered relative to each other.

Finally, it is likely that the systems' own cache policies will also place the data from the active working set into system cache, but that is outside the scope of this paper.

7.6 Discussion

In our experiments, we only allowed the system to cache fingerprints it had seen during the course of the trace. This emulates starting from a clean slate, but in a real system once it reaches a steady state the amount of data pulled in should increase.

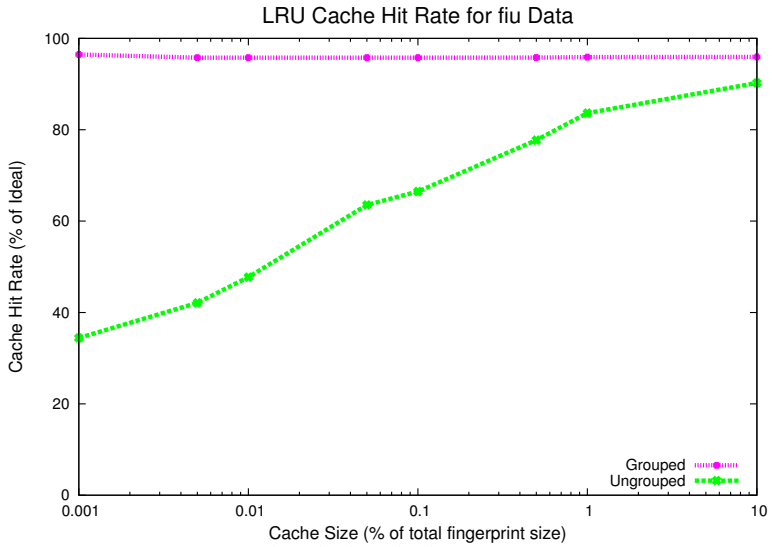
It is important to note that, contrary to our expectations, adding groups to the index cache never reduced the cache hit rate. This indicates that there is room for larger groups to be pulled in before cache churn starts becoming a serious problem. Thus, a workload specific implementation with domain informed grouping will have a greater improvement in cache hits. For example, a system with extra computational power could update groups continually, resulting in more large groups and corresponding cache hits.

We create groups based on a bare-bone set of features, namely location and timestamp. We used these to show our method was valuable even in cases where it is not practical to extract rich system traces. Additional features such as request origin or client type could make the groups more reflective of real workload phenomena. Any method for group prediction or even file level access prediction can be substituted into our system with little modification, making HANDS a general purpose tool to improve primary deduplication.

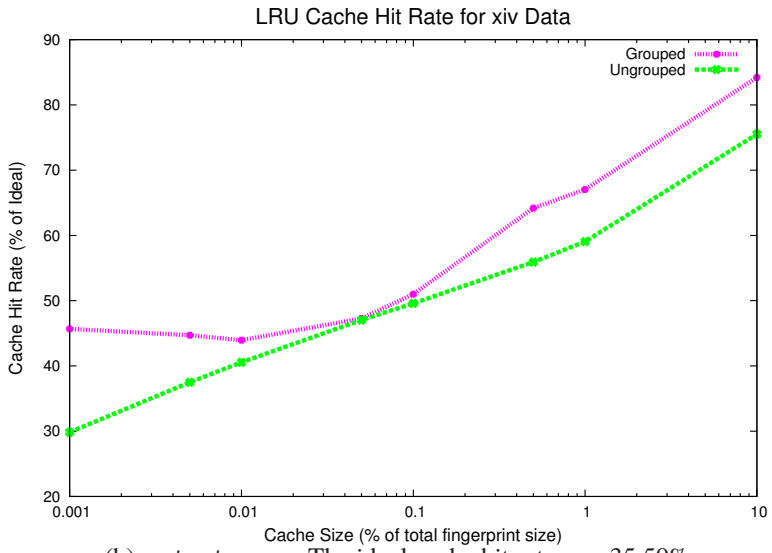
Though we showed significant improvement in research and enterprise deduplication, there are particular workloads where we believe HANDS will shine. For example, a collection of virtual machine images would have both higher deduplication ratios and tighter groups, since they correspond closely to individual systems. Workloads from businesses that operate based on strict timing rules such as banks and trading houses

should show a cyclic usage pattern that would be amenable to our deduplication method. In addition to large scale data, our technique could be applied to object stores where any reduction in media cost is amplified because of the relative expense of storage class memory technologies. Our method could also be used to cost-effectively address the growing problem of archival-like storage that does not obey "write-once, read maybe" semantics and instead has transient periods of primary activity. Storing exabytes of data and maintaining a usable primary deduplication index is prohibitive, but storing .1% of the index size in memory should be much more manageable and cost effective for long term storage such as Internet archives and media.

Our design translates directly into fewer disk accesses for inline deduplication and, from there, better user-facing performance. HANDS is highly modular and adaptable to specific environmental constraints, and thus is an approach that can be deployed in nearly any system to alleviate the disk bottleneck problem. Although neither of our workloads had the requisite 45% deduplication ratio to benefit from traditional primary deduplication, with HANDS both achieve near perfect deduplication using between .01% and 10% of the memory. This reduces the required deduplication ratio to break even on cost to a mere 8% of incoming data if using 4KB blocks. Since such a small percentage of fingerprints need to be kept in cache for good performance, we encourage wider adoption of primary deduplication in industry and even personal servers.

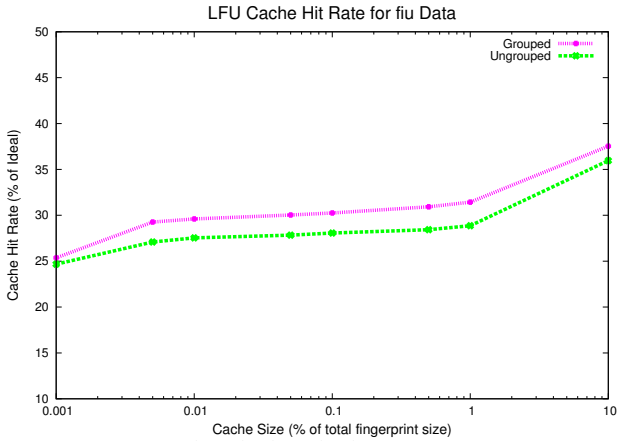


(a) `fiu`: The ideal cache hit rate was 33.94%

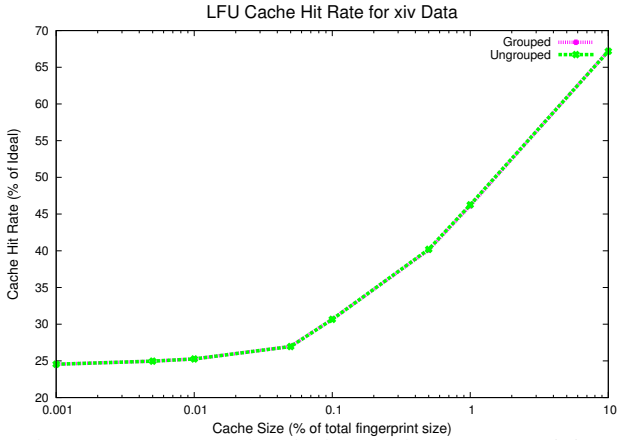


(b) `ent-storage`: The ideal cache hit rate was 35.59%

Figure 7.7: LRU Cache hits across cache sizes. Grouped data does consistently at least as well and often significantly better than ungrouped data.

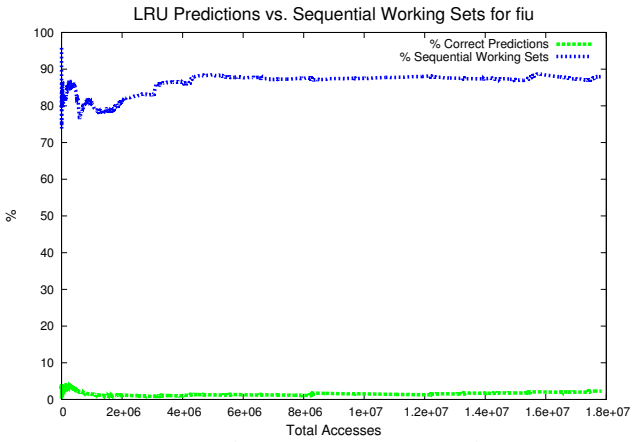


(a) `fiu`: The ideal cache hit rate was 33.94%

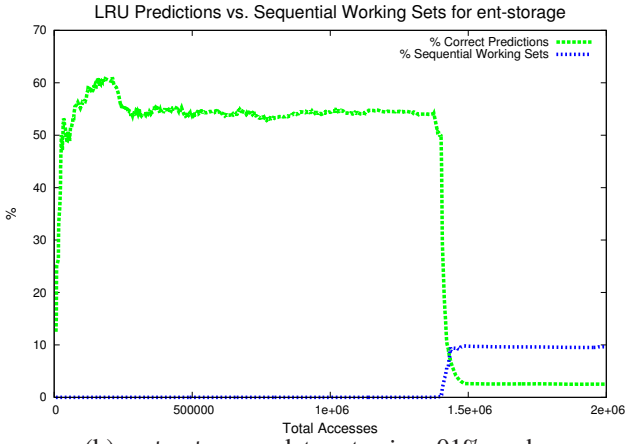


(b) `ent-storage`: The ideal cache hit rate was 35.59%

Figure 7.8: LFU Cache hits across cache sizes. The `ent-storage` dataset sees no benefit from grouped data while the `fiu` dataset sees a modest benefit from grouping.

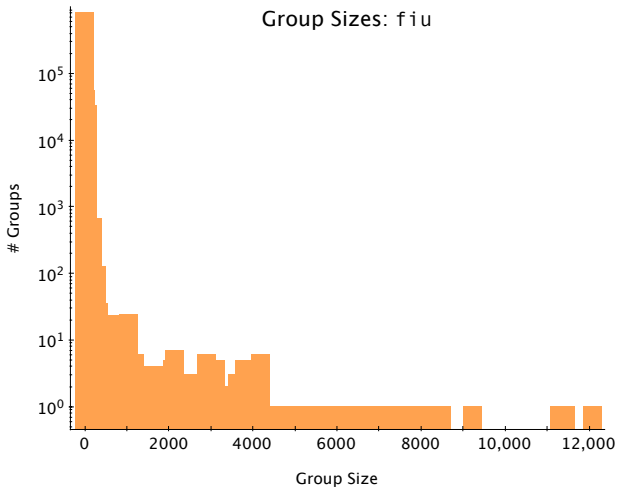


(a) `fiu` data set using .01% cache

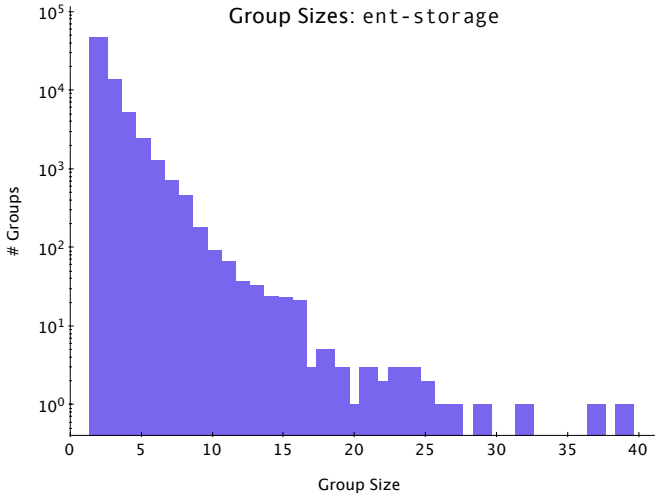


(b) `ent-storage` data set using .01% cache

Figure 7.9: For the `fiu` data set using LRU, predictive power of groups was unrelated to sequentiality. In the `ent-storage` data set using LRU, predictive power of groups fell as sequential groups increased. The percentage of predictive accesses is deceivingly low because it is calculated as a percentage of total accesses, which were an order of magnitude higher for `fiu` than `ent-storage`. The cache size was .01%



(a) **fiu**



(b) **ent-storage**

Figure 7.10: Distribution of group sizes for **fiu** and **ent-storage** grouped under NNP.

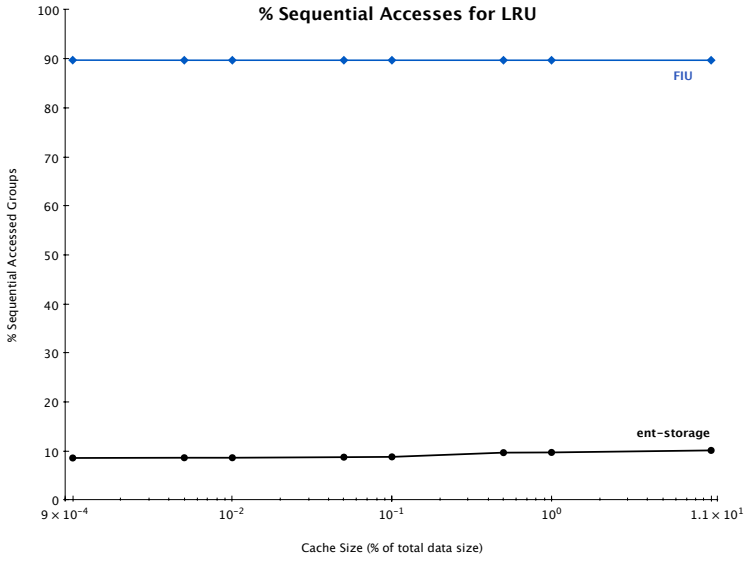


Figure 7.11: The `fiu` dataset has significantly more sequential groups accessed than the XIV dataset

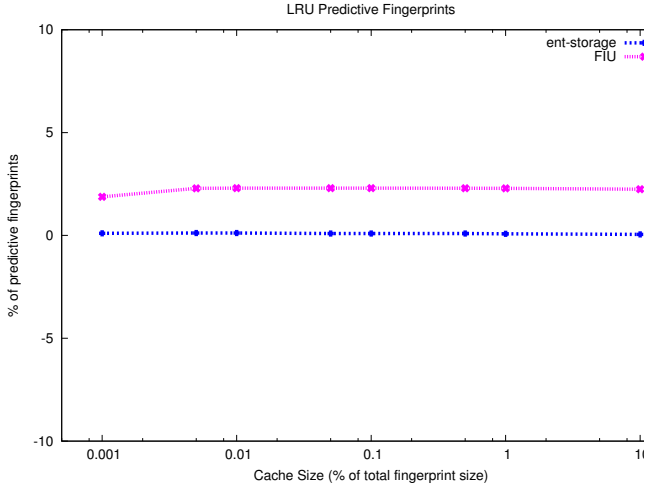


Figure 7.12: The `fiu` dataset has a slightly higher percentage of predictive group members than `ent-storage`

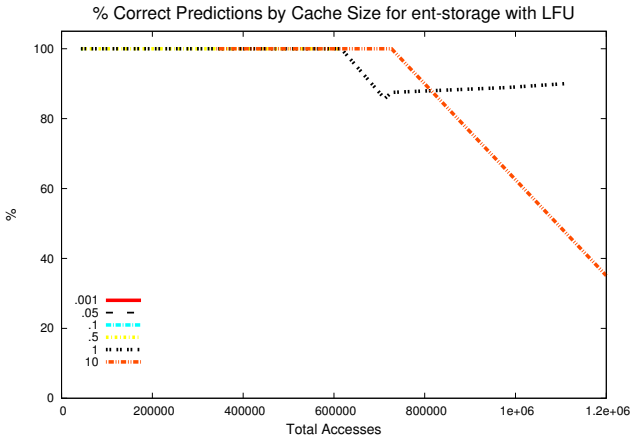
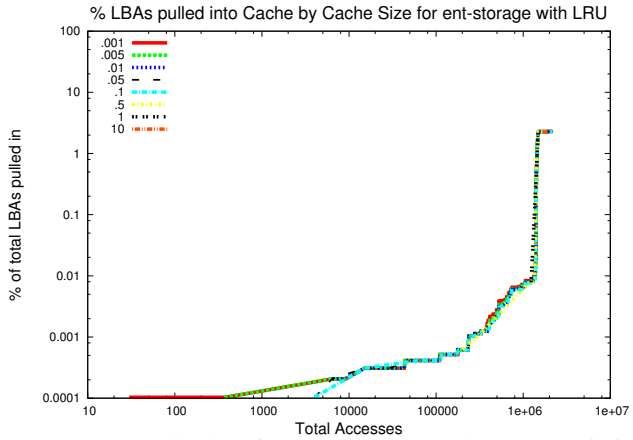
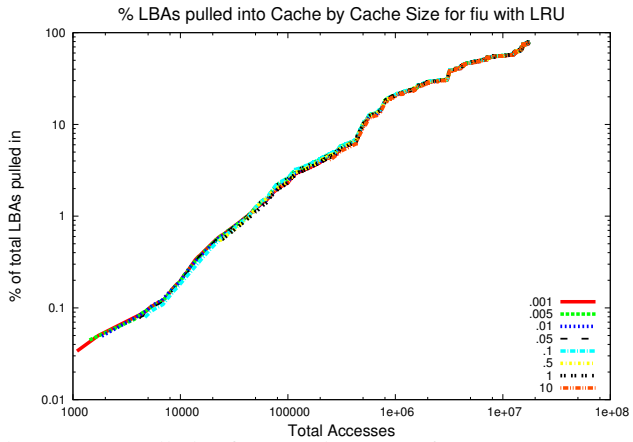


Figure 7.13: % of correct predictions by cache size, compared against the total fingerprints pulled into cache and not immediately replaced. We see that the few fingerprints that were pulled in were predictive at small cache sizes, while extra elements were pulled into a larger cache. Cache sizes below 1% had fewer predictions and are thus hidden by the overlap in the graph.



(a) % LBAs pulled in for `ent-storage`; there are 968620 unique LBAs



(b) % LBAs pulled in for `fiu`; there are 1684407 unique LBAs

Figure 7.14: Each line corresponds to a run with the given cache size. Adding groups to `ent-storage` with LRU achieves an increased cache hit rate while only pulling in $< 3\%$ of the total LBAs. Conversely, `fiu` with LRU pulls in up to 80%.

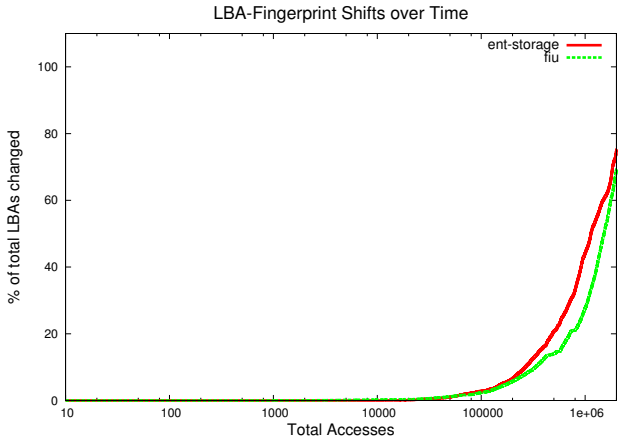
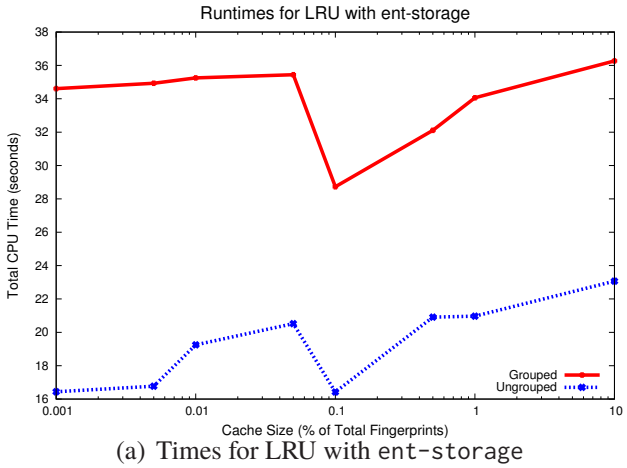
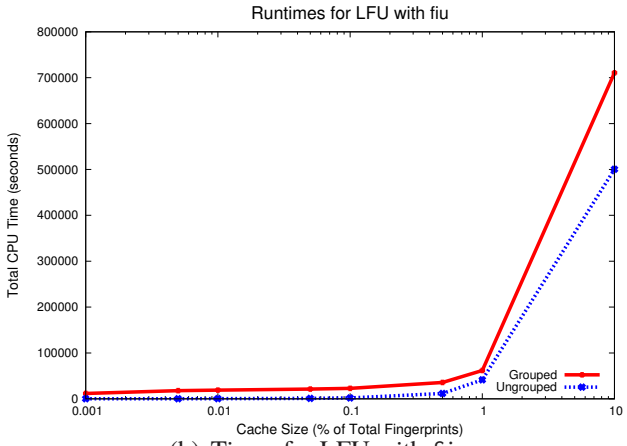


Figure 7.15: LBA-fingerprint correlation shifts over time. Though the data sets have very different characteristics, both show a sharp rise in shifts after about 500,000 accesses.



(a) Times for LRU with `ent-storage`



(b) Times for LFU with `fiu`

Figure 7.16: A comparison of runtime vs. cache size for our algorithms. This was run in a prototyping environment; for translation to a real system the key is that the grouped performance closely tracks the ungrouped performance with a fixed overhead.

Chapter 8

Discussion and Further Directions

> *I am an optimist. Anyone interested in the future has to be otherwise he would simply shoot himself.*
>
> **– Arthur C. Clarke**

Based on the work we have done to identify and apply groups to storage systems, we have identified a few promising directions for this research. In this chapter, we discuss our current work and future projects.

8.1 Choosing a Grouping Technique

The first task we faced in each of the previous three application areas was choosing a grouping technique from the several we have studied. Our grouping techniques produce groupings that are broadly either fast to recalculate or are very tightly fit to the training data. For instance, we designed NNP in response to a scenario where groups changed very quickly and any group calculation needed to be fast enough to respond to a changing workflow. In HANDS, we needed to maximize the predictive power of groups in a small, fixed size cache, so we further restricted NNP to bias heavily towards smaller groups. Power in an archival system, on the other hand, required larger groups to minimize the layout overhead, and these groups could be recalculated much less frequently based on our assumptions about the archival workload.

As a result of this experience, we can also broadly divide application areas as "grouping for layout" and "grouping for caching". We class an application as grouping for layout when the goal of the grouping is to lay out data on physical media such that co-located data has a high probability of co-access. We showed instances of this application type when using groups to reduce power consumption (Ch. 5) and improve availability (Ch. 6). In these scenarios, space is relatively plentiful but groups can not change quickly, since the change requires a layout overhead. Here, we want larger, more stable groups, such as those produced by our bag-of-edges technique, along with ideally a more stable workload.

Grouping for cache management, on the other hand, requires a grouping technique that biases towards smaller groups in order to avoid cache churn: when the additional data pulled in by the grouping is evicted before it has a chance to be useful. Along with smaller groups, groupings that populate caches should have parameter weights set to bias more strongly towards temporal correlations since the lifetime of the group in cache is so limited. If an application has a rapidly changing workload, a grouping technique such as neighborhood partitioning that has a strong bias towards newer correlations significantly outperforms grouping techniques that need more history. This is also why we set window size for NNP as a function of the incoming IOPS: NNP is designed to capture recent workload shifts.

8.2 Workload Characterization

Once we have more data, our next step is to discover what about a workload makes it amenable to this sort of grouping. We believe that workloads with distinct use cases, whether they be from an application or a user, are the best bet for future grouping efforts, but many HPC and long-term storage workloads share some of the surface level properties that make the application servers good candidates. The goal of this line of questioning is to derive a set of characteristics of a workload that would indicate how easy it is to group along with what parameters to try first.

Another angle we are interested in is backtracking from our working sets to discover which sources tend to access the same offsets of data. Once we know this, we can implement more informed cache prefetching and, in large systems, physically move the correlated offsets near to each other on disk to avoid unnecessary disk activity. Previous work has led us to believe that even if files are duplicated across disks, the potential gain from catching subsequent accesses in large, mostly idle systems is high enough to make it worthwhile [121]. We are also interested in refining the graph covering algorithm to accept groups that are only partially connected instead of requiring complete cliques.

8.3 Power Management for Replicated File Systems

Many storage systems for organizations that require very high availability handle reliability by replicating data across multiple, physically distant servers. The Google File System (GFS) is one such storage system that maintains at least three consistent replicas of data [90]. The GFS paper does not discuss how replicas are placed in detail, but there is no indication in the paper that replicas are placed based on a metric other than load balancing.

The PNUTS system at Yahoo! does arrange records by write locality. In particular, they periodically scan where a record is accessed from most and then strive to move the master to be geographically closer to the access point whereas replicas are put wherever they are best for net load balancing [20]. While this has the side effect of grouping codependently accessed data in the same geographic location, the focus is not to reduce disk accesses but to reduce latency.

Instead of having the replicas in arbitrary, load-balanced servers, these could be placed into working sets and the working sets could then be treated as record replicas currently are. We propose simulating a large scale storage environment with replication to determine the impact of placing replicas into working sets instead of placing them purely for load balancing. The motivation for this work arises from the realization that some pieces of data classify into multiple working sets with varying probability, and only

placing the data in the most likely working set is sub-optimal. In replicated systems, multiple copies of the data already exist, so we can create overlapping working sets without increasing the storage burden of the system.

The trade-off to placing replicas into working sets is expected to be an increase in latency. We believe that this increase will be manageable, especially in a hybrid scheme where both load balancing and working set placement are taken into account.

To determine the impact of grouping on replicated systems, it would be interesting to simulate systems with pre-defined levels of replication or a pre-defined level of reliability to be gained from a combination of replication and parity. The control could be a system simulated to load balance evenly across servers. This control could be placed against a system load balanced for write locality, similar to PNUTS, a system where all replicas are placed into working sets, and a hybrid system of working sets and load balancing. Another interesting tradeoff to explore is that between grouping with write locality versus read locality.

8.4 Isolating Correlated Failures

We could represent more failure scenarios under PERSES if we add correlated failures to our fault simulator. We expect PERSES to do well with correlated failures because there will be fewer, larger failures, which PERSES is designed for. Another interesting problem is in how groups are allocated to disks. Currently, disks are filled in with groups based on group size. We are exploring using more intelligent bin packing to place groups on disk based on probability of access. Finally, we are looking at the effects of combining PERSES with existing systems to reduce reconstruction overhead through replication and caching.

Our eventual goal is to design a data layout algorithm for non-hierarchical file systems. Current file systems use the directory hierarchy to obtain some notion of likelihood of co-access in data. If we can automatically detect projects and lay them out such that projects are isolated, we can control fragmentation in non-hierarchical systems with-

out administrative overhead or time consuming metadata analysis.

To optimally handle correlated failures, we also need to introduce the notion of group similarity, leading to a "meta-grouping" of groups. Analyzing meta-groups leads to a new class of problems in identification and group layout that are outside our current scope, but are essential for doing a meaningful analysis of correlated clustered behavior.

8.5 Grouping for SLO Management

As systems get larger and more progress is made on storage class memories such as memristors and phase change RAM, we predict a natural progression towards more tiered storage systems that contain many sets of data with different usage patterns and requirements. The question then is how do we allocate different data to different tiers of storage without having to manually specify correlations for every new file. Additionally, we need to treat files that are members of multiple usage groups in a way that meets the potentially conflicting expectations of each group.

One standard way to codify these expectations is to define service level objectives (SLOs) for different classes of data. SLOs are a means for an organization to allocate scarce resources among different focus areas. These objectives can then be enforced by service level agreements (SLAs) which add a cost model to the objectives. SLOs and their accompanying SLAs form the backbone of many companies' quality assurance. Questions include how best to identify which data falls under which SLO, how to efficiently move groups of data to a different storage tier, and how to re-assign different data elements to different tiers as the workload changes. Data grouping could answer all three of these. For example, by tying the SLOs to features, we can assign these groupings to given service levels without manually assigning individual files to SLOs.

We can also better manage SLOs through groupings. Part of the complexity of a good SLO implementation is bin packing: how best to fit sets of desired outcomes over a set of resources that deliver those outcomes with a given probability. By arranging groups defined by sets of features, we have more information about where to place entire

classes of data based on the observed level of service that some data receives. Finally, by having groupings update dynamically, we can quickly move data to the desired level of storage even as the workload changes. One interesting question here will be how restructuring file groups using external tagsets will affect the overall system reliability and power consumption. This could lead to a more realistic reward/loss function than storage systems with SLOs use today [115, 130].

Chapter 9

Conclusion

> *In the development of the understanding of complex phenomena, the most powerful tool available to the human intellect is abstraction. Abstraction arises from the recognition of similarities between certain objects, situations, or processes in the real world and the decision to concentrate on these similarities and to ignore, for the time being, their differences.*
>
> **– C.A.R. Hoare**

The thread that knits this thesis together is identifying and utilizing sets of data with high probability of co-access that do not necessarily have any externally discernible semantic relationship. In trying to understand storage access from a statistical perspective, we can move beyond metadata or popularity to find data that is accessed together due to unobserved secondary or tertiary effects. This allows our methodologies to generalize to a variety of emerging systems: our algorithms scale in effectiveness with the complexity of the system, as we can find patterns that someone with domain knowledge may not. Once we have these patterns, we can intelligently lay out data based on the desired application, creating an infinite, on-disk cache of related data.

The key insights we have discovered in this work are:

We can learn meaningful group correlations from block I/O data

In Chapter 4, we see that groups can be formed from historical trace data that con-

tains a minimal number of fields – we find relevant, persistent groups in 2-dimensional data. This shows that data grouping is possible even on systems with high performance requirements, where collecting metadata places undue overhead on a system, or systems where the privacy requirements make using semantic and metadata information to classify data untenable. Additionally, we can do this with little performance overhead.

We can calculate groups with high predictivity quickly and scalably

Our N-NP technique (Chapter 4.1.4.2) runs in $O(n)$, making it possible to recalculate groupings online, catching shifts in trace patterns in real time. By tracking group likelihood and predictivity, we manage to avoid being overzealous in regrouping data, instead finding a balance between past knowledge and new trace data. Finally, N-NP scales to workloads with arbitrary IOPS and to machines with arbitrary amounts of RAM to do the grouping calculation. Using a windowed technique for scalability allows us to be confident that group identification will be possible on many future workloads.

A grouped system has a better power profile than traditional MAID

As we saw in Chapter 5, the groups that we see in data help reduce the power overhead in a way that simple popularity can not. With relatively stable co-accessed sets such as those in archival data, we can save a significant amount of the power a system consumes under normal use by laying out data on disks according to group membership. This is impossible with popularity because data would be moved too often to redistribute popular data, causing additional disk spins that hurt the overall power footprint. Grouping, at least in archival workloads, solves the issue of MAID disk spinups reversing the power gains of leaving disks off.

Groups can isolate faults to fewer devices

Laying out data according to groups also allows us to reduce the impact of faults on different users and projects accessing a shared storage system. We showed that we can save over 9,000 hours, or over a year, of time compared to a random allocation that projects would otherwise spend waiting for a member to reconstruct over a three year trace by identifying and laying out projects based on statistical likelihood of co-access between blocks. We also show that grouped allocation saves time over an ideal temporal allocation of blocks onto disks, demonstrating that the grouping we find is catching subtle relationships that simple temporal correlation cannot.

Grouping can mitigate the disk-bottleneck problem in primary deduplication by pre-selecting fingerprints into the index cache

We also showed that group identification is useful for more than just laying data out on media. By their nature, groups guarantee a level of co-access, making them ideal for pre-fetching elements into a cache. This is particularly useful when, as is the case for primary deduplication, the grouping table can be held in memory. We showed in Chapter 7 that grouping fingerprints and then pre-fetching entire groups at once reduced the amount of in-memory cache required to catch most accesses by .1%. This is sufficient for many (though not all) real world workloads, and shows the promise of the grouping methodology outside of simple layout.

All of the results in this thesis were not positive. For example, we found that graph-theoretic methods based on clique cover overfit and were not capable of tracking large groups of correlated blocks. Additionally, we found that clustering methods such as k-means and EM could not handle interleaved groups, and so returned results that were indistinguishable from noise.

An additional weakness we see is that many of the applications we discuss depend

on disk. We believe that disks, or devices with similar characteristics, will be a major component of large or archival storage systems for the foreseeable future. That said, though the applications we look at rely on disk, there are viable potential applications for a group-based layout or input method in other storage media. For example, splitting groups up along flash chips could help with load balancing along different channels, and making deduplication more affordable has even higher impact on media with higher acquisition costs.

We are excited that some of the work in this thesis, particularly HANDS, is being implemented to make real systems more efficient. This research goes hand in hand with the realization that the lines of usage patterns between traditional archival and modern large scale, mixed use storage systems is blurring. By identifying working sets, we can predict high level system behavior to smooth out some of the haphazard nature of multi-user workloads.

Bibliography

[1] I.F. Adams, M.W. Storer, and E.L. Miller. Analysis of workload behavior in scientific and historical long-term data repositories. Technical report, Tech. Report UCSC-SSRC-11-01, UC Santa Cruz, 2011.

[2] I.F. Adams, M.W. Storer, and E.L. Miller. Analysis of workload behavior in scientific and historical long-term data repositories. *ACM Transactions on Storage (TOS)*, 8(2):6, 2012.

[3] A. Amer, D.D.E. Long, J.F. Paris, and R.C. Burns. File access prediction with adjustable accuracy. In *IPCCC 2002*, pages 131–140. IEEE Computer Society, 2002.

[4] H. Amur, J. Cipar, V. Gupta, G.R. Ganger, M.A. Kozuch, and K. Schwan. Robust and flexible power-proportional storage. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 217–228. ACM, 2010.

[5] I. Ari, A. Amer, R. Gramacy, E.L. Miller, S.A. Brandt, and D.D.E. Long. ACME: adaptive caching using multiple experts. In *Proceedings in Informatics*, volume 14, pages 143–158. Citeseer, 2002.

[6] A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, L.N. Bairavasundaram, T.E. Denehy, F.I. Popovici, V. Prabhakaran, and M. Sivathanu. Semantically-smart disk systems: past, present, and future. *ACM SIGMETRICS Performance Evaluation Review*, 33(4):29–35, 2006.

[7] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An analysis of latent sector errors in disk drives. In *Proceedings of the 2007 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 2007.

[8] M. Baker, K. Keeton, and S. Martin. Why traditional storage systems donâĂŹt help us save stuff forever. In *HotDep 2005*, pages 2005–120, 2005.

[9] M. Barbaro and T. Zeller Jr. A face is exposed for aol searcher no. 4417749. August 2006.

[10] L.A. Barroso and U. Holzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[11] D. Bhagwat, K. Eshghi, D.D.E. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009. MASCOTS'09. IEEE International Symposium on*, pages 1–9. IEEE, 2009.

[12] A. Borodin and R. El-Yaniv. *Online computation and competitive analysis*, volume 53. Cambridge University Press Cambridge, 1998.

[13] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *IEEE INFOCOM'99*, volume 1, 1999.

[14] E. V. Carrera, E. Pinheiro, and R. Bianchini. Conserving disk energy in network servers. In *ICS '03*, pages 86–97, 2003.

[15] S. Chen and D. Towsley. The design and evaluation of RAID 5 and parity striping disk array architectures. *Journal of Parallel and Distributed Computing*, 17(1-2):58–74, 1993.

[16] E. G. Coffman, Jr. and Thomas A. Ryan, Jr. A study of storage partitioning using a mathematical model of locality. *Commun. ACM*, 15(3):185–190, March 1972.

[17] D. Colarelli and D. Grunwald. Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, page 11. IEEE Computer Society Press, 2002.

[18] C. Constantinescu, J. Glider, and D. Chambliss. Mixing deduplication and compression on active data sets. In *2011 Data Compression Conference*, pages 393–402. IEEE, 2011.

[19] P. Constantopoulos, M. Doerr, and M. Petraki. Reliability modelling for long term digital preservation. In *9th DELOS Network of Excellence thematic workshop "Digital Repositories: Interoperability and Common Services", Foundation for Research and Technology-Hellas (FORTH)*. Citeseer, 2005.

[20] B.F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[21] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[22] W. Dai. Crypto++ 5.6.0 benchmarks. 2009.

[23] P. Desnoyers, D. Ganesan, and P. Shenoy. Tsar: A two tier sensor storage architecture using interval skip graphs. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, page 50. ACM, 2005.

[24] A. Dholakia, E. Eleftheriou, I. Iliadis, J. Menon, and KK Rao. Analysis of a new intra-disk redundancy scheme for high-reliability RAID storage systems in the presence of unrecoverable errors. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 373–374. ACM New York, NY, USA, 2006.

[25] Western Digital. Western digital caviar green specification sheet. Western Digital, 2010. `http://www.wdc.com/wdproducts/library/?id=132&type=8`.

[26] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang. DiskSeen: exploiting disk layout and access history to enhance I/O prefetch. In *2007 USENIX ATC*, pages 1–14. USENIX Association, 2007.

[27] S. Doraimani and A. Iamnitchi. File grouping for scientific data management: lessons from experimenting with real traces. In *Proceedings of the 17th international symposium on High performance distributed computing*, pages 153–164. ACM, 2008.

[28] F. Douglis, P. Krishnan, and B. Bershad. Adaptive disk spin-down policies for mobile computers. *Computing Systems*, 8(4):381–413, 1995.

[29] F. Douglis, P. Krishnan, and B. Marsh. Thwarting the power-hungry disk. In *1994 USENIX ATC*, page 23. USENIX Association, 1994.

[30] R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern classification*, volume 2. Citeseer, 2001.

[31] The Economist. Data, data everywhere. *The Economist Newspaper Limited*, February 2010.

[32] P. Efstathopoulos and F. Guo. Rethinking deduplication scalability. In *HotStorageâĂŹ10, 2nd Workshop on Hot Topics in Storage and File Systems*, 2010.

[33] D. Essary and A. Amer. Predictive data grouping: Defining the bounds of energy and latency reduction through predictive data grouping and replication. *Trans. Storage*, 4(1):1–23, 2008.

[34] B. Dufrasne et al. *IBM XIV Storage System Gen3 Architecture, Implementation, and Usage*. IBM, International Technical Support Organization.

[35] L. Ganesh, H. Weatherspoon, M. Balakrishnan, and K. Birman. Optimizing power consumption in large scale storage systems. In *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, page 9. USENIX Association, 2007.

[36] J.F. Gantz, C. Chute, A. Manfrediz, S. Minton, D. Reinsel, W. Schlichting, et al. The Diverse and Exploding Digital Universe. *IDC White Paper*, 2, 2008.

[37] F. Gens. Idc predictions 2012: Competing for 2020. *IDC, Report, Dec*, 2011.

[38] T. Gibson and E.L. Miller. An improved long-term file usage prediction algorithm. In *Proceedings of the 25th International Conference for the Resource Management and Performance and Performance Evaluation of Enterprise Computing Systems (CMG99)*, pages 639–648, Reno, NV, December 1999.

[39] K. Gopinath, N. Muppalaneni, N.S. Kumar, and P. Risbood. A 3-tier RAID storage system with RAID1, RAID5 and compressed RAID5 for Linux. In *2000 USENIX ATC*, page 30. USENIX Association, 2000.

[40] K.M. Greenan, D.D.E. Long, E.L. Miller, SJ Schwarz, and J.J. Wylie. A spin-up saved is energy earned: achieving power-efficient, erasure-coded storage. In *Proceedings of the Fourth conference on Hot topics in system dependability*, pages 4–4. USENIX Association, 2008.

[41] K.M. Greenan, E.L. Miller, T.J.E. Schwarz, and D.D.E. Long. Disaster recovery codes: increasing reliability with large-stripe erasure correcting codes. In *Proceedings of the 2007 ACM workshop on Storage security and survivability*, pages 31–36. ACM, 2007.

[42] I. Gupta, T.D. Chandra, and G.S. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 170–179. ACM, 2001.

[43] D.P. Helmbold, D.D.E. Long, T.L. Sconyers, and B. Sherrod. Adaptive disk spin-down for mobile computers. *Mobile Networks and Applications*, 5(4):285–297, 2000.

[44] R.Y. Hou, J. Menon, and YN Patt. Balancing i/o response time and disk rebuild

time in a raid5 disk array. In *System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on*, volume 1, pages 70–79. IEEE, 1993.

[45] P. Jaccard. Distribution de la flore alpine dans le bassin des dranses et dans quelques rÃľgions voisines. *Bulletin del la SociÃľtÃľ Vaudoise des Sciences Naturelles*, 37:241–272, 1901.

[46] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality. In *FAST 2005*, page 8. USENIX Association, 2005.

[47] K. Jin and E. L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, May 2009.

[48] L. Jones, M. Reid, M. Unangst, and B. Welch. Panasas tiered parity architecture. *Panasas White Paper*, 2008.

[49] S. Jones. Online de-duplication in a log-structured file system for primary storage. Technical Report UCSC-SSRC-11-03, University of California, Santa Cruz, May 2011.

[50] D. Jurafsky and J.H. Martin. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition*. Prentice Hall, 1st edition, 2000.

[51] J.M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, , and C.S. Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, 2000.

[52] R. Koller and R. Rangaswami. I/o deduplication: utilizing content similarity to improve i/o performance. *ACM Transactions on Storage (TOS)*, 6(3):1–26, 2010.

[53] T.M. Kroeger and D. D.E. Long. Design and implementation of a predictive file prefetching algorithm. In *USENIX Annual Technical Conference, General Track*, pages 105–118, 2001.

[54] T.M. Kroeger and D.D.E. Long. Predicting file system actions from prior events. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, page 26. Usenix Association, 1996.

[55] M.H. Kryder and C.S. Kim. After hard drives - what comes next? *Magnetics, IEEE Transactions on*, 45(10):3406–3413, 2009.

[56] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, et al. Oceanstore: An architecture for global-scale persistent storage. *ACM SIGARCH Computer Architecture News*, 28(5):190–201, 2000.

[57] H. Lamehamedi, Z. Shentu, B. Szymanski, and E. Deelman. Simulation of dynamic data replication strategies in data grids. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 10–pp. IEEE, 2003.

[58] Katherine Lawrence. Re-thinking the lamp stack: Part 2. *PINGV*, December 2010.

[59] Y. H. Lee. To defrag or not to defrag âĂŞ- that is the question for ssd. 2008.

[60] A. Leventhal. Triple-parity raid and beyond. *Queue*, 7(11):30, 2009.

[61] Z. Li, Z. Chen, S.M. Srinivasan, and Y. Zhou. C-miner: Mining block correlations in storage systems. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 173–186. USENIX Association, 2004.

[62] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proceedings of the 7th conference on File and storage technologies*, pages 111–123. USENIX Association, 2009.

[63] Louwrentius. Raid array size and rebuild speed. 2010.

[64] D.S.H. Rosenthal M. Roussopoulos P. Maniatis TJ Giuli M. Baker, M. Shah and P. Bungale. A fresh look at the reliability of long-term digital storage. In *Proceedings of EuroSys 2006*, pages 221–234, April 2006.

[65] S. J. Leffler M. K. McKusick, W. N. Joy and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[66] M. K. McKusick M. Seltzer, K. Bostic and C. Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 307–326, January 1993.

[67] A.E. Magurran. Measuring biological diversity. 2004.

[68] T.J. Mahoney. The long tail internet myth: Top 10 domains arenâĂŹt shrinking. Compete Inc., 2006. `http://blog.compete.com/2006/12/19/long-%20tail-chris-anderson-top-10-domains/`.

[69] N. Mandagere, P. Zhou, M.A. Smith, and S. Uttamchandani. Demystifying data deduplication. In *Proceedings of the ACM/IFIP/USENIX Middleware'08 Conference Companion*, pages 12–17. ACM, 2008.

[70] D. McCullagh. Why no one cares about privacy anymore. 2010. `http://news.cnet.com/8301-13578_3-20000336-38.html`.

[71] J. Metz. Working document of the new technologies file system (ntfs).

[72] D.T. Meyer and W.J. Bolosky. A study of practical deduplication. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, pages 1–1. USENIX Association, 2011.

[73] G. Grider J. Gemmill J. Harris M.L. Curry, H.L. Ward and D. Martinez. Power use of disk subsystems in supercomputers. In *Proceedings of the sixth workshop on Parallel Data Storage (PDSW'11)*, pages 49–54. ACM, 2011.

[74] N. Muppalaneni and K. Gopinath. A multi-tier RAID storage system with RAID1 and RAID5. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 663–671, 2000.

[75] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):1–23, 2008.

[76] J. Oly and D.A. Reed. Markov model prediction of I/O requests for scientific applications. In *Proceedings of the 16th international conference on Supercomputing*, pages 147–155. ACM, 2002.

[77] J.F. Pâris, A. Amer, and D. D. E. Long. A stochastic approach to file access prediction. In *The International Workshop on Storage Network Architecture and Parall I/Os (SNAPI '03)*, sep 2003.

[78] A. Parker-Wood, B. Madden, M. McThrow, D.D.E. Long, I.F. Adams, and A. Wildani. Examining extended and scientific metadata for scalable index designs. In *Systor 2013*, June 2013.

[79] A. Parker-Wood, C. Strong, E.L. Miller, and D.D.E. Long. Security aware partitioning for efficient file system search. May 2010.

[80] E. Pinheiro and R. Bianchini. Energy conservation techniques for disk array-based servers. In *ICS '04*, pages 68–78. ACM, 2004.

[81] E. Pinheiro, W.D. Weber, and L.A. Barroso. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*, 2007.

[82] E. L. Miller Q. Xin and T. J. E. Schwarz. Evaluation of distributed recovery in large-scale storage systems. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 172–181, Honolulu, HI, June 2004.

[83] T. J.E. Schwarz D. D. E. Long S. A. Brandt Q. Xin, E. L. Miller and W. Litwin. Reliability mechanisms for very large storage systems. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 146–156, April 2003.

[84] L. Tan Y. Zhou K. Keeton Q. Zhu, Z. Chen and J. Wilkes. Hibernator: Helping disk arrays sleep through the winter. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, UK, October 2005. ACM.

[85] L. Alvisi R. Kotla and M. Dahlin. SafeStore: a durable and practical storage system. In *Proceedings of the 2007 USENIX Annual Technical Conference*, pages 129–142, June 2007.

[86] W.M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66:846–850, 1971.

[87] A. Rigo and S. Pedroni. Pypy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 944–953. ACM, 2006.

[88] A. Riska and E. Riedel. Disk drive level workload characterization. In *Proceedings of the USENIX Annual Technical Conference*, pages 97–103, 2006.

[89] J.P. Rybczynski, D.D.E. Long, and A. Amer. Adapting predictions and workloads for power management. In *MASCOTS 2006*, pages 3–12, 2006.

[90] H. Gobioff S. Ghemawat and S. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, October 2003. ACM.

[91] P. B. Gibbons S. Nath, H. Yu and S. Seshan. Subtleties in tolerating correlated failures in wide-area storage systems. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[92] B. Satzger, A. Pietzowski, W. Trumler, and T. Ungerer. A new adaptive accrual failure detector for dependable distributed systems. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 551–555. ACM, 2007.

[93] J. Schindler, J.L. Griffin, C.R. Lumb, and G.R. Ganger. Track-aligned extents: matching access patterns to disk drive characteristics. In *Conference on File and Storage Technologies*, 2002.

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

[95] B. Schroeder, S. Damouras, and P. Gill. Understanding latent sector errors and how to protect against them. *ACM Transactions on Storage (TOS)*, 6(3):9, 2010.

[96] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–16, February 2007.

[97] B. Schroeder and G.A. Gibson. Understanding failures in petascale computers. In *Journal of Physics: Conference Series*, volume 78, page 012022. IOP Publishing, 2007.

[98] T. J. E. Schwarz, Q. Xin, E. L. Miller, D. D. E. Long, A. Hospodor, and S. Ng. Disk scrubbing in large archival storage systems. In *Proceedings of the 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '04)*, pages 409–418, October 2004.

[99] T. Schwarz, S. J. and E. L. Miller. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *Proceedings of the 26th International Conference on Distributed Computing Systems (ICDCS '06)*, Lisboa, Portugal, July 2006. IEEE.

[100] M. Sivathanu, V. Prabhakaran, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau. Improving storage system availability with D-GRAID. *ACM TOS*, 1(2):133–170, 2005.

[101] M. Sivathanu, V. Prabhakaran, F.I. Popovici, T.E. Denehy, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 73–88, 2003.

[102] N. Slonim, G. Singh Atwal, G. Tkacik, and W. Bialek. Information-based clustering. *Proceedings of the National Academy of Science*, 1021:18297–18302, December 2005.

[103] T. Sø renson. A method of establishing groups of equal amplitude in plant sociology based oil similarity of species content. *K. danske Videnskab.Selsk. biol.Skr*, pages 1–34, 1948.

[104] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proccedings of the 10th conference on File and storage technologies*. USENIX Association, 2012.

[105] K. Sripanidkulchai, B. Maggs, and H. Zhang. An analysis of live streaming workloads on the internet. In *ACM SIGCOMM conference on Internet measurement*, pages 41–54. ACM, 2004.

[106] C. Staelin and H. Garcia-Molina. Clustering active disk data to improve disk performance. *Princeton, NJ, USA, Tech. Rep. CS–TR–298–90*, 1990.

[107] M.W. Storer, K. Greenan, D.D.E. Long, and E.L. Miller. Secure data deduplication. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, pages 1–10. ACM, 2008.

[108] M.W. Storer, K.M. Greenan, E.L. Miller, and K. Voruganti. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *6th USENIX FAST*, pages 1–16. USENIX Association, 2008.

[109] X.H. Sun, Y. Chen, and Y. Yin. Data layout optimization for petascale file systems. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, pages 11–15. ACM, 2009.

[110] A.S. Tanenbaum, J.N. Herder, and H. Bos. File size distribution on UNIX systems: then and now. *ACM SIGOPS Operating Systems Review*, 40(1):104, 2006.

[111] A. Thomasian, Y. Tang, and Y. Hu. Hierarchical raid: Design, performance, reliability, and recovery. *Journal of Parallel and Distributed Computing*, 2012.

[112] L. Tiair, H. Jiang, D. Feng, H.Q. Xin, and X. Shir. Implementation and evaluation of a popularity-based reconstruction optimization algorithm in availability-oriented disk arrays. In *Mass Storage Systems and Technologies, 2007. MSST 2007. 24th IEEE Conference on*, pages 233–238. IEEE, 2007.

[113] Y. Tsuchiya and T. Watanabe. Dblk: Deduplication for primary block storage. In *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, pages 1–5. IEEE, 2011.

[114] J. Urquhart. Does the fourth amendment cover 'the cloud'? 2008. `http://news.cnet.com/8301-19413_3-10436425-240.html`.

[115] S. Uttamchandani, K. Voruganti, R. Routray, L. Yin, A. Singh, and B. Yolken. BRAHMA: Planning Tool for Providing Storage Management as a Service. In *Proceedings of the IEEE International Conference on Services Computing*, pages 1–10, 2007.

[116] P. van Gelder, G. Beijer, and M. Berger. Statistical analysis of pageviews on web sites. *Information and Communication Technologies*, 28, 2002.

[117] J. Wang and Y. Hu. Profs-performance-oriented data reorganization for log-structured file system on multi-zone disks. In *mascots*, page 0285. Published by the IEEE Computer Society, 2001.

[118] J. Wang, H. Zhu, and D. Li. eraid: Conserving energy in conventional disk-based raid system. *IEEE Transactions on Computers*, pages 359–374, 2007.

[119] C. Weddle, M. Oldham, J. Qian, A. Wang, P. Reiher, and G. Kuenning. PARAID : A gear-shifting power-aware RAID. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, February 2007.

[120] S.A. Weil, S.A. Brandt, E.L. Miller, D.D.E. Long, and C. Maltzahn. Ceph: a scalable, high-performance distributed file system. In *OSDI 2006*, page 320. USENIX Association, 2006.

[121] A. Wildani and E.L. Miller. Semantic data placement for power management in archival storage. In *Petascale Data Storage Workshop (PDSW), 2010 5th*, pages 1–5. IEEE, 2010.

[122] A Wildani, E.L. Miller, and O. Rodeh. Hands: A heuristically arranged non-backup in-line deduplication system. Technical Report UCSC-SSRC-12-03, University of California, Santa Cruz, March 2012.

[123] A. Wildani, E.L. Miller, and L. Ward. Efficiently identifying working sets in block i/o streams. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, page 5, 2011.

[124] A. Wildani, T.J.E. Schwarz, E.L. Miller, and D.D.E. Long. Protecting against rare event failures in archival systems. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009. MASCOTS'09. IEEE International Symposium on*, pages 1–11. IEEE, 2009.

[125] R. Wood. Future hard disk drive systems. *Journal of magnetism and magnetic materials*, 321(6):555–561, 2009.

[126] S. Wu, H. Jiang, D. Feng, L. Tian, and B. Mao. Workout: I/o workload outsourcing for boosting raid reconstruction performance. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FASTâĂŹ09)*, 2009.

[127] J.J. Wylie, M. Bakkaloglu, V. Pandurangan, M.W. Bigrigg, S. Oguz, K. Tew, C. Williams, G.R. Ganger, and P.K. Khosla. *Selecting the right data distribution scheme for a survivable storage system*. School of Computer Science, Carnegie Mellon University, 2001.

[128] A. Veitch A. Merchant Y. Saito, S. Frolund and S. Spence. FAB: Building distributed enterprise disk arrays from commodity components. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 48–58, 2004.

[129] T. Yang, H. Jiang, D. Feng, Z. Niu, K. Zhou, and Y. Wan. Debar: A scalable high-performance de-duplication storage system for backup and archiving. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[130] L. Yin, S. Uttamchandani, M. Korupolu, K. Voruganti, and R. Katz. SMART: an integrated multi-action advisor for storage systems. In *2006 USENIX ATC*, pages 229–242, 2006.

[131] S. Zaman, S.I. Lippman, L. Schneper, N. Slonim, and J.R. Broach. Glucose regulates transcription in yeast through a network of signaling pathways. *Molecular Systems Biology*, 5(1), 2009.

[132] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, page 18. USENIX Association, 2008.

[133] X. Zhuang and H.H.S. Lee. Reducing cache pollution via dynamic data prefetch filtering. *IEEE Transactions on Computers*, pages 18–31, 2007.