Title

Can We Group Storage? Statistical Techniques to Identify Predictive Groupings in Storage System Accesses

Journal

ACM Transactions on Storage, 12(2)

ISSN

1553-3077

Authors

Wildani, Avani
Miller, Ethan L

Publication Date

2016-03-08

DOI

10.1145/2738042

Peer reviewed

Can We Group Storage? Statistical Techniques to Identify Predictive Groupings in Storage System Accesses

AVANI WILDANI, The Salk Institute for Biological Studies
ETHAN L. MILLER, University of California, Santa Cruz

Storing large amounts of data for different users has become the new normal in a modern distributed cloud storage environment. Storing data successfully requires a balance of availability, reliability, cost, and performance. Typically, systems design for this balance with minimal information about the data that will pass through them. We propose a series of methods to derive groupings from data that have predictive value, informing layout decisions for data on disk.

Unlike previous grouping work, we focus on dynamically identifying groupings in data that can be gathered from active systems in real time with minimal impact using spatiotemporal locality. We outline several techniques we have developed and discuss how we select particular techniques for particular workloads and application domains. 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 domain 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.

Categories and Subject Descriptors: I.5.1 [**Pattern Recognition**]: Models; D.4.2 [**Operating Systems**]: Storage Management

General Terms: Design, Algorithms, Measurement, Performance

Additional Key Words and Phrases: Data layout, storage optimization, tiered storage, predictive modeling

ACM Reference Format:
Avani Wildani and Ethan L. Miller. 2016. Can we group storage? Statistical techniques to identify predictive groupings in storage system accesses. ACM Trans. Storage 12, 2, Article 7 (February 2016), 33 pages.
DOI: http://dx.doi.org/10.1145/2738042

1. INTRODUCTION

A pressing issue in systems is management of "big data," data that is too massive to immediately process, which leads to results that themselves are nontrivial to process and store. Moreover, 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 multiuser, multiapplication system must appear as a dedicated server to each user.

This research was also supported in part by the National Science Foundation under awards CNS-0917396 (part of the American Recovery and Reinvestment Act of 2009 [Public Law 111-5]) and IIP-0934401, and by the Department of Energy's Petascale Data Storage Institute under award DE-FC02-06ER25768. We also thank Sandia National Laboratories and the industrial sponsors of the Storage Systems Research Center and the Center for Research in Intelligent Storage for their generous support.
Authors' addresses: A. Wildani, Dept. of Math & CS, Emory University, 400 Dowman Dr., W401, Atlanta, GA 30322; email: avani@mathcs.emory.edu; E. L. Miller, Computer Science Department, Baskin School of Engineering, University of California, 1156 High Street, MS SOE3; email: elm@soe.ucsc.edu.

DOI: http://dx.doi.org/10.1145/2738042

This article approaches the problem of availability and power requirements of big data on large, heterogeneous systems by identifying groups within the data that share a high probability of coaccess to approximate a dedicated server for different users or applications through informed cache population and data layout. Similar grouping in specialized systems has been shown to help avoid crossing track boundaries [Schindler et al. 2002], isolate faults [Sivathanu et al. 2005], and avoid power consumption from excessive disk activity [Pinheiro and Bianchini 2004]. The main contributions of our grouping methods are adaptability and domain independence. By examining the basic statistics of the training workloads, we can quickly adjust our model to changes in workload characteristics.

We define a *group* as any set of storage units, such as blocks, files, or objects, that are likely to be accessed together within a defined amount of time, which is typically a function of the rate of storage requests. Consequently, any element of a group acts as a predictor for other elements in that group. Groups typically arise from working sets of user or application data, though they can also represent higher-level correlations such as interapplication dependencies and operating system interactions.

In an ideal world, we would be able to group objects based on either data contents of the application, user, or use case typical for that object. We term this sort of information-driven labeling *categorical grouping*. Categorical grouping is a well-studied problem [Arpaci-Dusseau et al. 2006], and thus for datasets with rich metadata, we can treat the grouping problem as mostly solved and focus on datasets that lack this metadata.

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, even in the absence of any metadata. We discuss a variety of techniques for finding groups in this data 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 IOPS 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 rearranged lazily.

The end goal of this work is to predict future access probabilities based on prior information to the extent that these access patterns can inform scalable system design. Once groups exist, we can place them on the same types of physical media, prefetch elements a group at a time, or even structure reliability schemes to minimize intergroup availability disruptions. We have tested some of our techniques to address system power consumption through group-based layout and idle disk spin-down [Wildani and Miller 2010], and we have also had positive results using groups to populate a memory-based index cache for online deduplication [Wildani et al. 2013].

After reviewing other work in the field, we present our statistical similarity metrics and partitioning algorithms, along with a discussion of how parameters are selected and which methodology is best suited to which type of workload. We then present the working set groupings that each of these partitioning algorithms returns. We finally discuss the validity of our groupings and conclude with a discussion of the implications of our groupings and the work we are currently doing to both improve groupings and characterize workloads based on ability to be grouped into working sets.

2. BACKGROUND AND RELATED WORK

Grouping data for performance gains and other optimizations has a long history. The original BSD FFS aimed to localize associated data and metadata blocks within the same cylinder groups to avoid excessive seeks [McKusick and Fabry 1984].

Subsequent projects have focused on specific workloads or applications where grouping could provide a benefit. The DGRAID project demonstrated that placing semantically related blocks of a file adjacent to each other on a disk reduced the system impact on failure, since adjacent blocks tend to fail together [Sivathanu et al. 2005]. Localizing the failure to a specific file reduces the number of files that have to be retrieved from backups. 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. 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 [Schindler et al. 2002]. 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 under 3,000 bytes [Tanenbaum et al. 2006], we believe our technique can be used to help define track-aligned extents. Our end goal is to tease apart the accesses instigated by separate applications in order to obtain sets of blocks that are likely to be read or written to together. In one of our datasets, we find that the read:write ratio is almost 10:90, implying that our workload is directly comparable to the workload for personal computers with single disks in Riska's workload characterization study [Riska and Riedel 2006]. Riska also suggests the idea of using a protocol analyzer to collect I/O data without impacting the underlying system, opening block I/O analysis to active HPC systems that cannot tolerate any performance degradation for tracing.

Arpaci-Dusseau et al. have made a variety of advances in semantically aware disk systems [Sivathanu et al. 2003; Arpaci-Dusseau et al. 2006]. Their online inference mechanism had trouble with the asynchronous nature of modern operating systems. We use a longer history of block associations to uncouple the relationships between applications, and we are working on implementing their inference techniques as a secondary classification layer. Their techniques for inode inference and block association gain a great amount of information by querying the blocks; however, there is an implicit assumption here that we can identify and parse the metadata. We collect only the bare minimum of data, which allows our algorithms to work almost domain blind.

Dorimani and Iamnitchi [2008] discuss a need for characterization of HPC workloads for the purpose of file grouping. They also demonstrate a grouping using static, prelabeled groups, where the mean group size is about an order of magnitude larger than the mean file size. Prelabeled 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 interaccess similarity, we hope to be able to form dynamic groups from real-time access data. Oly and Reed [2002] present a Markov model to predict I/O requests in scientific applications. By focusing on scientific applications, their work bypasses the issue of interleaved groups. Yadwadkar et al. [2010] also use Markov modeling, and they apply their model to NFS traces, doing best when groups are not interleaved. Their method is more difficult to adapt to online data than the algorithm we present.

Essary and Amer [2008] provide a strong theoretical framework for power savings by dynamically grouping blocks nearby on a disk. Other predictive methods have shown good results by offering the choice of "no prediction," allowing a predictor to signal uncertainty in the prediction [Amer et al. 2002]. C-Miner uses frequent sequence matching on block I/O data, using a global frequent sequence database [Li et al. 2004]. Frequent sequence matching is susceptible to interlaced working sets within data and thus best for more specialized workloads, whereas our technique is suitable for multiapplication systems.

2.1. Grouping Versus Caching and Prefetching

Cache prefetching and clustering active disk data exploit the fact that recently accessed data is more likely to be accessed again in the near future on a typical server [Staelin and Garcia-Molina 1990]. Unlike several techniques that group data based on popularity or "hotness," we group data by likelihood of contemporaneous and related access regardless of the likelihood for the group, or any of its members, to be accessed at all. We also present partitioning algorithms including graph theoretic techniques that have not yet been considered for predictive grouping.

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 cache criteria picked based on what provides that best hit rate [Ari et al. 2002]. The cache criteria can involve file or block grouping [Pinheiro and Bianchini 2004], but typically only in the context of grouping together popular or hot blocks of the system [Wang and Hu 2001]. This is necessary because cache space is precious, so placing related, but less accessed data into the cache would only serve to pollute it [Zhuang and Lee 2007]. DULO biases the cache toward elements that have low spatial locality, increasing program throughput, but is affected by cache pollution issues for data that is rarely accessed [Jiang et al. 2005].

Our work is strongly based on previous work in cache prefetching techniques that predict file system actions based on previous events. Kroeger and Long [1996, 2001] examined using a variant of frequent sequence matching to prefetch files into cache. Their work provides strong evidence that some workloads (Sprite 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 [Narayanan et al. 2008].

Minimizing disk activity for disk accesses is especially important on some types of systems such as MAID where data is distributed around mostly idle disks [Colarelli and Grunwald 2002]. Diskseen performs prefetching at the level of the disk layout using a combination of history and sequence matching [Ding et al. 2007]. Pinheiro and Bianchini [2004] group active data together on disk to minimize the total number of disk spin-ups, but they are vulnerable to workloads where several blocks are typically accessed together but accessed infrequently. In a large system for long-term storage, the effect of these infrequent accesses can accumulate to be a large drain on power [Wildani and Miller 2010]. Recent studies have investigated using temporal locality [Lo et al. 2014] or spatial locality [Wu and He 2012] to more intelligently manage the flash translation in solid-state devices. The grouping methodologies we discuss have been tested for a variety of use cases including selectively prefetching fingerprints into a cache [Wildani et al. 2013], which may be a useful starting point for a generalized flash grouping technique.

2.2. Statistical Grouping Versus Categorical Grouping

Departing from previous work, we pay more attention to statistical grouping over categorical. Recent studies indicate that categorical grouping does not have the flexibility necessary for modern, shifting workloads [Wildani et al. 2014]. One silver lining of the massive amounts of data that 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.

Categorical grouping by definition requires some functional knowledge of the data that relies on human curation, either in manual labeling or metadata upkeep [Adams et al. 2012]. Example categorical attributes include size, name, type, owner, path, or even whole file content. 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 [Adams et al. 2012]. Also, if administrators leave, 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 multiuser, multipurpose storage systems, are constantly shifting. This makes it almost impossible to develop a generalizable technique to derive groups that have long-term predictive capability based on categorical data.

2.3. Grouping Versus 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 datasets we have observed thus far is strongly biased toward 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 shows 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 nonlinear relations [Slonim et al. 2005]. iClust has been used mainly in biology applications to cluster genes by expression data such that members of a cluster have high codependency [Zaman et al. 2009]. Though we did not use iClust to calculate the groupings for our applications because it is still much slower than NNP (Section 3.2.2), we believe that for a large, static system it would outperform bag-of-edges on more volatile data.

3. DESIGN

Grouping data is the necessary first step before we can explore colocation for power savings, fault isolation, avoiding track boundaries [Schindler et al. 2002], 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.

Modern storage systems are shifting from individual, low IOPS deployments to large shared storage servers such as Amazon's S3 that are under constant load. The prevalence of these systems is 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 [Constantinescu et al. 2011]. 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 prefetching, 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.

Our statistical techniques are designed to create groups quickly to adapt to changing conditions, using dynamically updated likelihood values and periodic regrouping based on performance, all 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 [Coffman and Ryan 1972].

Another reason we found that this was a better way to do trace-based prediction is privacy concerns, where organizations do not want to release data that could identify the users or applications that created a trace. 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 [Barbaro and Jr. 2006], 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 that is 5 to 10 years out of date, if they use real data at all [Adams et al. 2012].

The statistical grouping techniques we have researched all use data that can be collected nonintrusively 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 [Riska and Riedel 2006].

3.1. 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.

Statistical grouping requires data with at least two dimensions: time and something that can serve as an identifier as well as provide additional locality. The additional locality has the benefit of keeping our groupings resistant to noisy changes: for example, if a block offset is used as a UID, we can incorporate the bias in initial placement that offset indicates. 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.

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 (Figure 1), there is enough information in the offset to provide predictivity. This could in part be because the usage of data can remain similar even when parts of a file are overwritten.

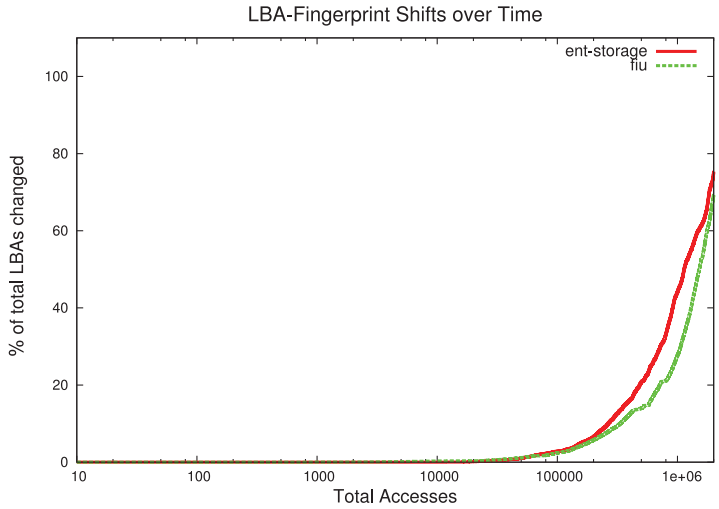


Fig. 1. An example of offset-data mismatches over time for a university (**fiu**) and an enterprise (**ent-storage**) dataset. Logical block addresses are used in place of physical offsets, and each LBA corresponds to a data fingerprint. In the enterprise dataset, **ent-storage**, labels remain consistent for many accesses before the percentage changed creeps up, while in the research university dataset represented by **fiu**, the mapping was more volatile. We recalculated 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.

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 1, the spatial dimension may map well to the data for some time. Making this assumption allows us to use very sparse data for our analysis, since spatiotemporal 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.

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. With more dimensions, we have a wide array of statistical similarity metrics available such as the Sørensen similarity index [Sørenson 1948], which biases against outliers, and Tanimoto Distance [Jaccard 1901], which provides a set comparison that is optimized for grouping seemingly dissimilar sets [Magurran 2004].

Distance

Our partitioning algorithms depend on a precalculated 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. We believe this is a result of controller- and OS-level prefetching techniques that decorrelate the size parameter from the working set. 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 toward 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 10 datapoints scattered throughout the system that are accessed, repeatedly, within a second of each other than we do about 10 datapoints that are adjacent on disk but accessed at random times over the course of our trace. For most of our grouping methods, we use a precalculated 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 and o = the unique ID dimension, and *oscale* is an IOPS-dependent weighting factor on the UID. As IOPS increases, the information in the current location of blocks decreases (Figure 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 3 shows average group sizes across the entire parameter space for a particular n-neighborhood partitioning grouping (Section 3.2.2). It shows that for a given workload, only a small range of *oscale* values produce a nontrivial 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 dataset. 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. Note that this is more complex than a straightforward Hamming distance because offsets occur multiple times within a trace window. 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

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

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 toward 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 3.2.2 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, $rDist(o_{i1}) = [(o_j, d(o_{i1}, o_j)), (o_{j+1}, d(o_{i1}, o_{j+1}), \ldots]$. 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 afterward. 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.

3.2. Statistical Partitioning Algorithms

The goal of all of the group partitioning algorithms we work with is to identify groups that have a high probability of coaccess 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 a definitive answer for the question "How far is offset a from offset b?" With this similarity information precomputed, we now look at the actual grouping of accesses into working sets.

3.2.1. Neighborhood Partitioning/N-Neighborhood Partitioning. Neighborhood partitioning (NP) is an online, agglomerative technique for picking groups in multidimensional 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. N-Neighborhood Partitioning (NNP) is a variation that improves scalability for dynamic grouping by dividing the stream into overlapping windows and combining resultant groupings. We do not use ranged or leveled distance lists with this method to limit computational overhead.

The partitioning steps for NP are as follows:

(1) Collect data.
(2) Calculate the pairwise distance matrix.
(3) Calculate the neighborhood threshold and sequentially detect groups in the I/O stream.
(4) Update likelihood values based on group reappearance.
(5) Drop groups below a likelihood threshold.

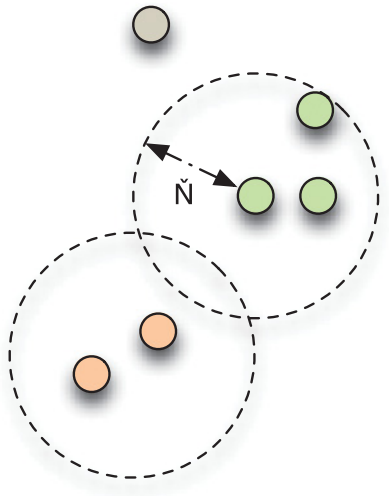


Fig. 2. Each incoming access is compared to the preceding access to determine whether it falls within the neighborhood ($\check{N}$) to be in the same group. If it does not, a new group is formed with the incoming access.

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 recalculated 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 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.

For our write-heavy, research dataset, we found that neighborhood partitioning was very susceptible to small fluctuations of its initial parameters and to the spike of writes in our workload. The modifications made to neighborhood partitioning to handle the high IOPs dataset (Section 3.2.2) fixes many of these issues to produce a consistent grouping across more parameter values.

Neighborhood partitioning runs in $O(n)$, where n is the size of the neighborhood, since it only needs to pass through each neighborhood twice: once to calculate the neighborhood threshold and again to collect the working sets. Once these passes are made, the cumulative symmetric difference can be calculated in $O(G)$, where G is the number of groups [Li 2008]. In our tests, we observed $G \ll n$, and so drop the term. Intuitively, this runtime makes sense because there is no $n \times n$ comparison step in NP; it is simply a linear agglomerator in a constrained window. This makes it an attractive grouping mechanism for workloads with high IOPS (e.g., the enterprise system we worked with can

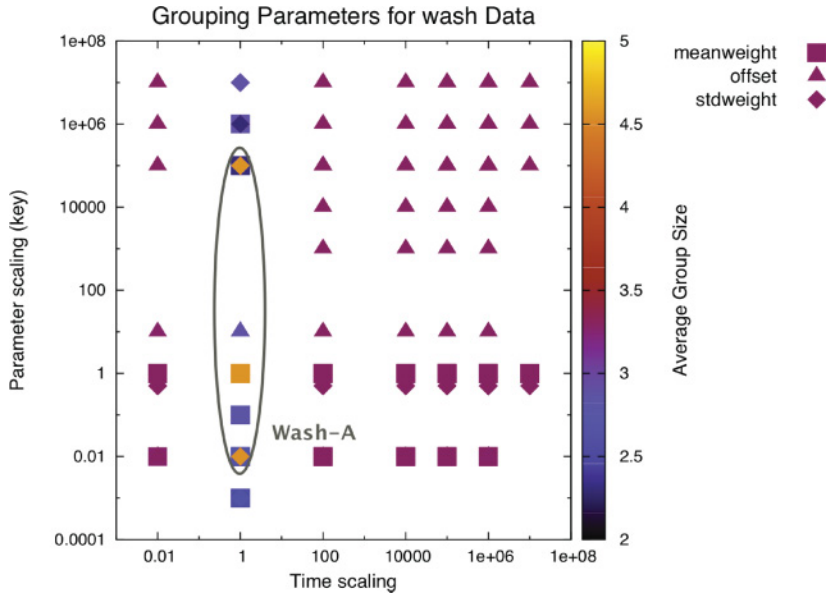


Fig. 3. Parameter selection for the Washington dataset with Neighborhood Partitioning. Color indicates the average group size produced by the specified parameters. Most parameter combinations result in grouping with the same average group size: a bit over 3. All of the datasets we tested showed similar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

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 3, we can weight our grouping parameters to bias NP toward 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 recalculation to prevent false negatives as the workload shifts, potentially negating their short-term predictive benefit.

3.2.2. 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 [Wildani et al. 2011]. 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.

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 an online deduplication scenario, we selected $w = 250$MB because of local memory constraints when populating the cache. 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, and moreover it is only updated when the predictivity of the overall grouping falls below a threshold. 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. The windows overlap by twice the current average group size to limit overcounting.

For NNP, for each window, the partitioning steps are as follows:

(1) Collect data.
(2) Perform neighborhood partitioning.
(3) Combine the new grouping with any prior groupings.
(4) Adjust likelihood of current groups.

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. Each group g_i has members $x_{i1}, x_{i2}, \ldots, x_{in}$. 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.

3.2.3. 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 [Duda et al. 2001]. 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, which is a group of size 44,000 that was consistently identified by both the k-NN and bag-of-edges algorithms. Similar effects occur with a background process doing periodic disk accesses.

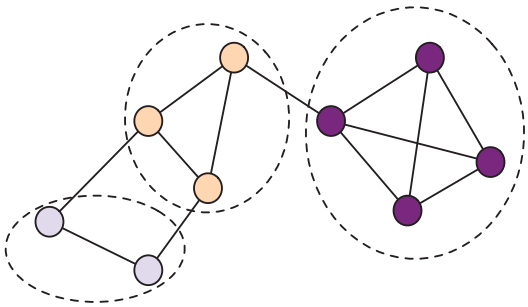


Fig. 4. A clique cover of a graph of accesses. Nodes represent accesses, while edges represent at least a threshold level of similarity between edges.

In our algorithm, we start with an $m \times m$ distance matrix as defined in Section 3.1, 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 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.

3.2.4. Graph Covering. The final 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 shows an example clique covering of an access graph. Note that every element is a member of a single working set that 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 [Cormen et al. 1990]. 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 dataset to find 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, and so forth, 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, our expected runtime is $\sum_2^x |G_i|^i$, where $|G_i|$ is the number of groups of size i. We observe in Figure 6 that the average group size tends to be small, and our runtime results in Section 5.3 show that this holds in real traces.

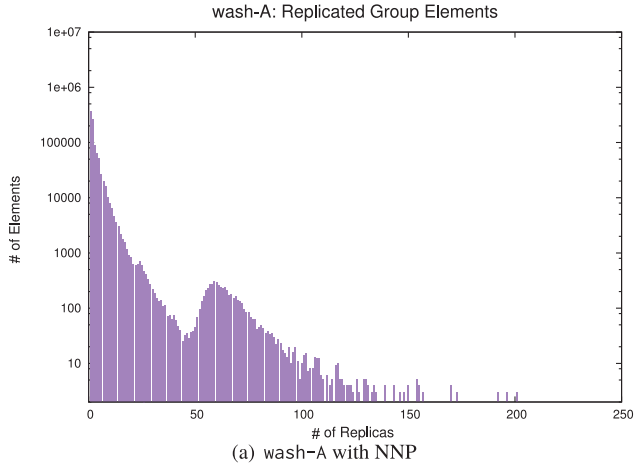
We discovered that in typical workloads, this method is too strict to discover most groups. This is likely because the accesses within a working set are the result of an ordered process. This implies that while the accesses will likely occur within a given range, the first and last access in the set 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 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 [Cormen et al. 1990]. 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, this technique offered the best combination of accuracy and performance, but this grouping algorithm was much too slow for dynamic grouping of high IOPS workloads.

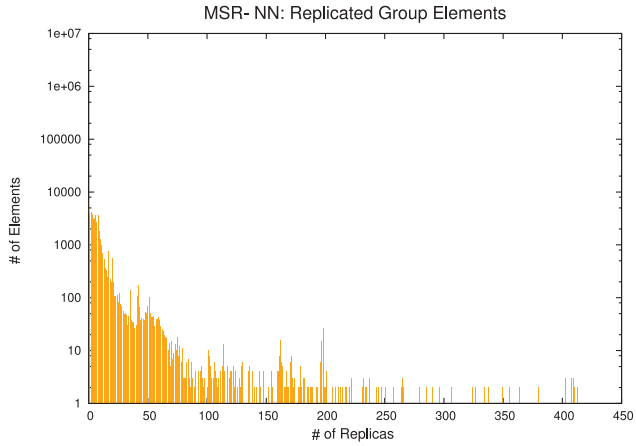
3.3. Group Likelihood and Predictivity

Group recalculation for all of the algorithms in the previous section happens in the background during periods of low activity. As accesses come in, however, we need to also proactively 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. Groups below a certain likelihood threshold can be discarded, though we only do this when there is an external limit on the number of groups (such as when the group table is being stored in memory) since these groups tend to be dropped during the periodic background regrouping. 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 5 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 overclassifying elements instead of labeling them as "noise."



(a) wash-A with NNP

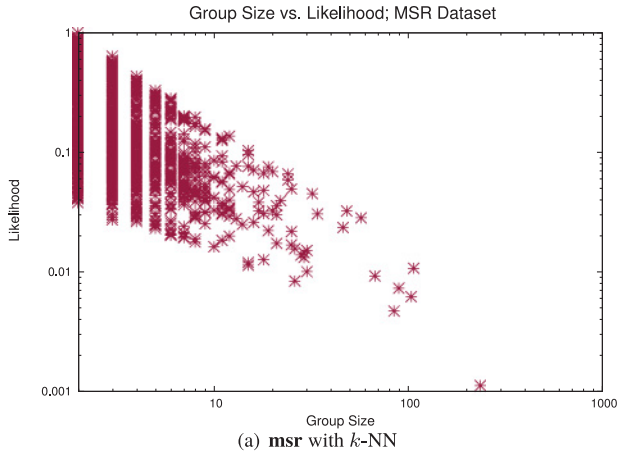


(b) **msr** with k-NN

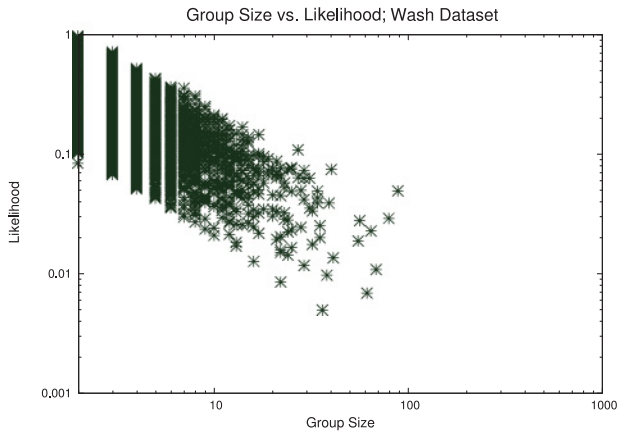
Fig. 5. Replicated elements for two statistical groupings.

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 is typically inversely correlated with group size, as we see in Figure 6. This serves to bias NNP toward 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 bag-of-edges 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 datasets that contain extra parameters such as initiator ID and process ID that



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



(b) **wash** with NNP

Fig. 6. In both NNP and k-NN, the most likely groupings tended to be smaller across all of our datasets. These graphs represent a subsample of 10% of groups per grouping, randomly selected. Likelihood is normalized within each grouping. These graphs show how even a small difference in average group size has a great impact on the grouping quality.

corroborate statistically derived groupings. Using this data, we verified that stable groups in these datasets are also products of the same process or initiator more than 90% of the time (Section 5.5).

We expect groups to be interleaved, so we cannot just check for the same groups to occur in many traces. Instead, we use the Rand index [Rand 1971], 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 that it does not penalize groups for the types 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

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

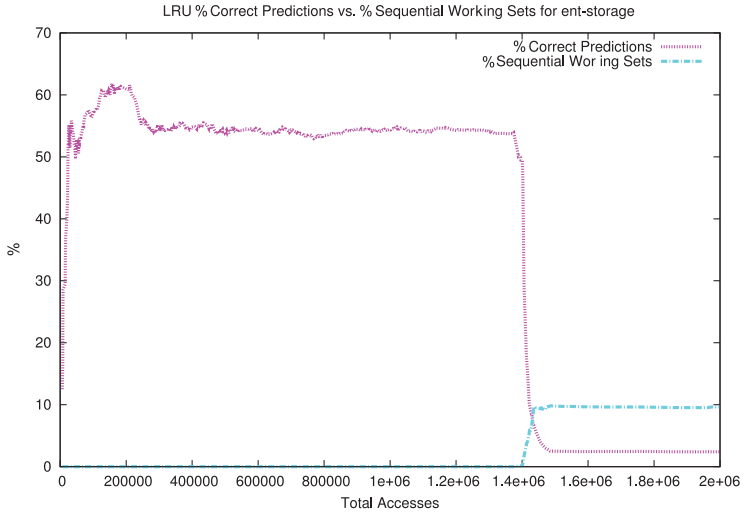


Fig. 7. Predictivity over time for NNP groups from the **ent-storage** dataset.

Table I. 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
xiv	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
wash	436MB	File	2007-10	100/0	5346868	39.70%	0.085	114	945 days

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.

Depending on the expected use of grouped data, the best test will be to run a large enough dataset 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 datasets, we can do actual verification on the predictive power of our grouping selections. Figure 7 shows the predictive power of a grouping for an enterprise dataset. We see that the predictive capability decreases after a certain point but is fixed by regrouping based on recent accesses. The frequency of regrouping depends on both the dynamism of the workload and the tolerances of the system. A system with little data ingress that has a very stable workload will not experience the loss in predictivity we see in Figure 7, whereas a system that is highly dynamic will see group predictivity fall quickly. At that point, the choice of when to regroup is determined by system usage. For example, the system could be set to regroup when predictivity falls below 10% but also regroup no more than once every 10,000 accesses. In our sample traces, we ended up regrouping about once every 100,000 accesses or not at all. Our Python prototype could typically regroup data in under a minute (see Table VI).

4. EXPERIMENTS

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 5 or even 10 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 that 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.

Our first dataset, **msr**, which we use for all of the algorithms we outline in this article, represents 1 week of block I/O traces from multipurpose enterprise servers used by researchers at Microsoft Research (MSR), Cambridge [Narayanan et al. 2008]. 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. Second, these traces were collected in 2007, making them more recent than most other publicly available block I/O traces. The offsets accessed in our data, which is from a single disk, were spaced between 581632 and 18136895488.

This dataset was very write heavy with a read/write ratio of 10:90. This ratio is almost entirely attributable to a small set of offsets that are likely to represent an on-disk cache, which is an anecdotally known feature of NTFS [Metz 2012]. Figure 8 shows the accesses by block over time, and Figure 8(b) highlights the read activity. Removing the writes (Figure 8(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.

Our second trace, **fiu**, is from Florida International University (FIU) and traces researchers' local storage [Koller and Rangaswami 2010]. This is a multiuser, 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.

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 4,096 bytes. We use LBA as the spatial component of the calculation. As of 2014, this data is publicly available at http://sylab.cs.fiu.edu/doku.php?id=projects:iodedup:start.

This dataset was also surprisingly write heavy, likely due to small system writes replicated per user (we obviously cannot know the actual cause). Over 33% of accesses were to duplicate blocks, determined by the MD5 hashes. 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-ahead, and 7TB of SSD cache [Dufrasne et al. 2012]. The directories are stored under GPFS [Schmuck and Haskin 2002] and are used by over 100 researchers.

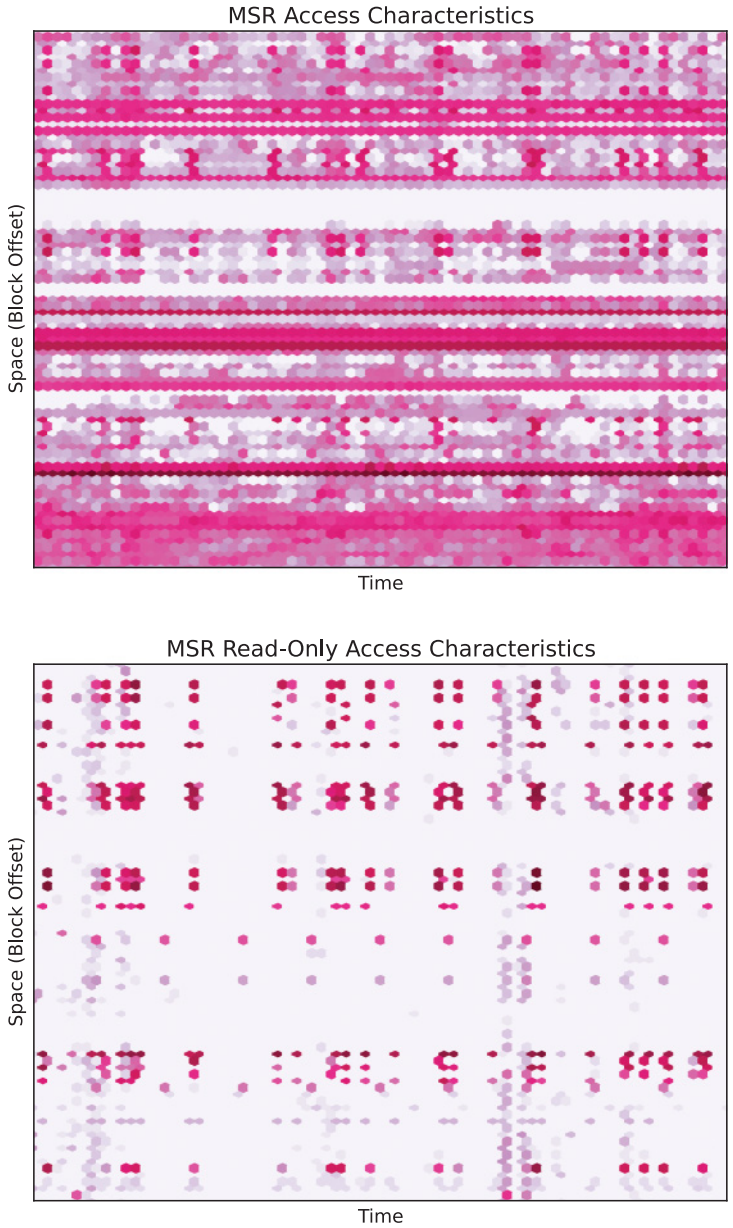


Fig. 8. These 2D histograms show the spatiotemporal layout of block accesses across the **msr** trace with and without writes. Darker bins correspond to higher access density.

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 postadaptive lookahead. This means that sequential accesses are effectively removed from the trace by the system itself. This dataset is not publicly available.

Our file-archive dataset is a database of vital records from the Washington state digital archives, where records are labeled with one of many type identifiers (e.g., "Birth Records," "Marriage Records") [Adams et al. 2012]. We examined 5,321,692

Table II. 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

Table III. 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...

Table IV. 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...

accesses from 2007 through 2010 that were made to a 16.5TB 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.

In addition to the access trace in Table V, we also had a file that mapped most of the RecordID values to assorted RecordType values such as "BirthRecord," "MarraigeRecord," and so forth. We treat RecordType as a prelabeled 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.

5. RESULTS

We ran every algorithm with **msr** and show that NNP provides consistently predictive groupings while bag-of-edges is the most stable. We also show the remainder of our datasets under NNP to demonstrate the versatility of the algorithm.

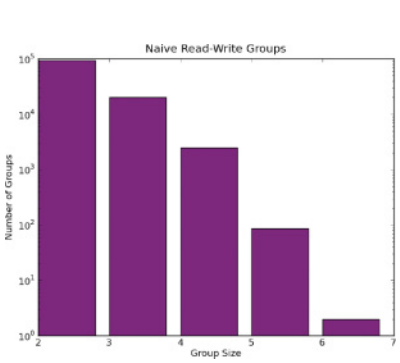
5.1. Neighborhood Partitioning

We tested the neighborhood partitioning algorithm on our data first 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. Neighborhood partitioning ended up being very susceptible to small fluctuations of its initial parameters and to the spike of writes in our workload. Figure 9 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 9(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

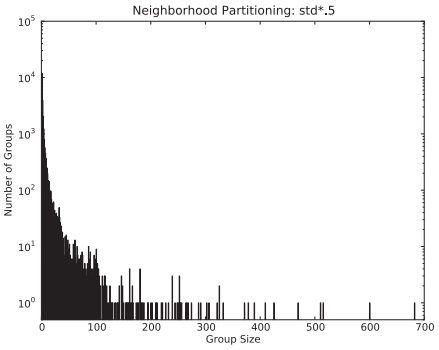
[1]This is not quite true, but accurate for our purposes. Further explanation can be found in Adams et al. [2012].

Table V. Sample Data from Washington Department of Records

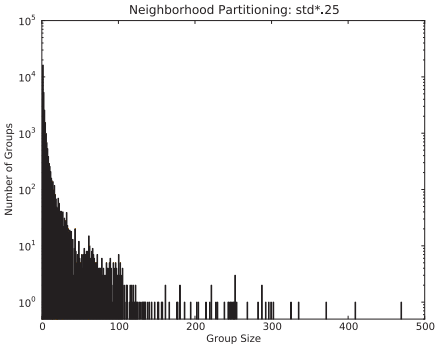
RetrieveTrackingID	UserSessionID	RecordID	RetrieveDate
1	{C8E99715-4725-427A-BCDF-708109D4935F}	34358	2007-09-27 13:31:10.407
2	{D2B7A983-7CC6-46C8-A10F-7B2557CF204F}	94267	2007-09-27 15:36:13.287
3	{1CE276B9-06F4-4AF7-9A08-E4038D83BBFB}	46679	2007-09-27 15:59:42.737



(a) All Accesses



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



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

Fig. 9. Working sets with Neighborhood Partitioning in **msr** for different values of stdweight (the weighting of the standard deviation). Groupings vary drastically based on neighborhood size and workload density.

(Figure 9(c)) causes the number of large groups to fall sharply and correspondingly increases the prevalence of small groups.

5.2. *k*-NN

Figure 10 shows the working sets returned by running k-NN with k ranging from 3,200 to 25,600. 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. 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 10 are calculated after the cache group is taken out. The cache group is a group of size 44,000 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. 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.

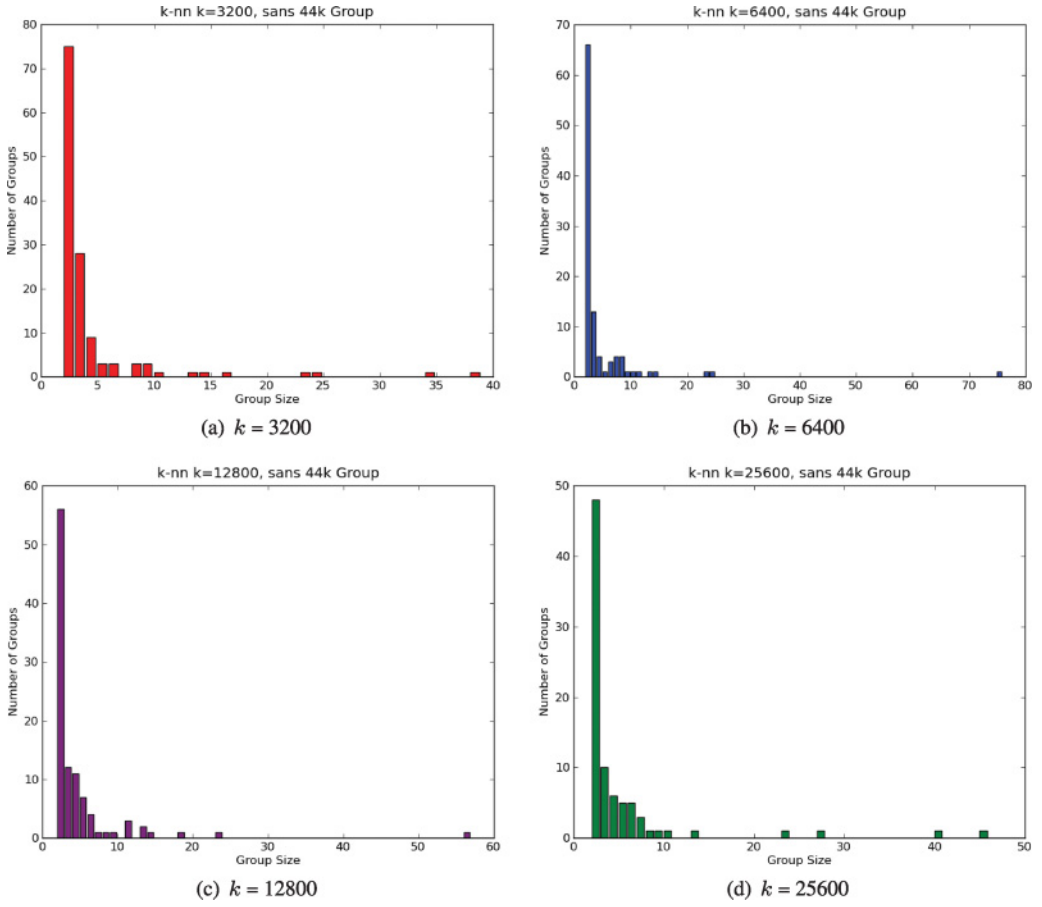


Fig. 10. Working sets with k-Nearest Neighbor. If k is very high or low, fewer large groups are found.

5.3. Bag-of-Edges

In the **msr** data, our clique-based graph algorithm failed to ever find a group above size 2. 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; that is, 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 11 shows the groupings bag-of-edges returns. The levels in Figure 11 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

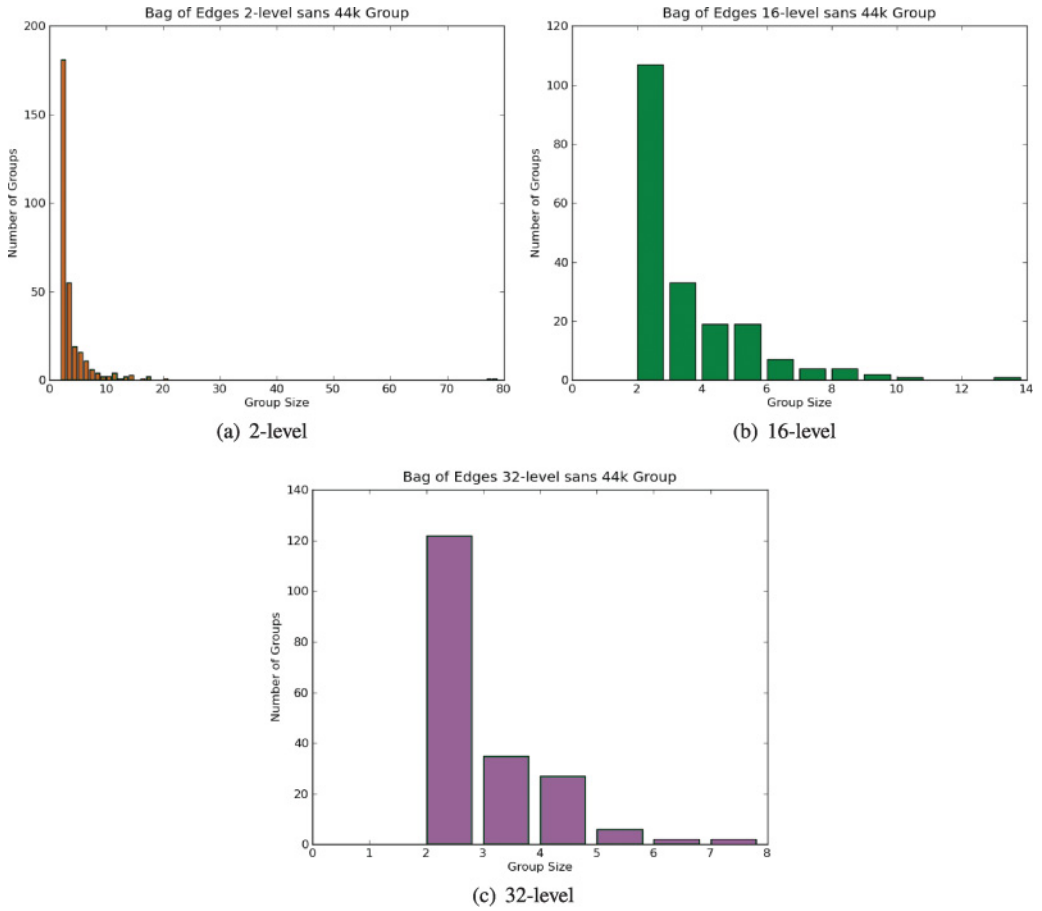


Fig. 11. Working sets with the bag-of-edges algorithm. Higher levels result in much smaller groupings.

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.

Figure 12 provides an example of the stability of bag-of-edges under varying the weighting factors added to the parameters that make up the distance metric: namely, time and the difference in offset numbers.

5.4. NNP

The majority of the applications we have investigated were run with *N*-Neighborhood Partitioning, where we saw promising results in grouping workloads ranging from fingerprints for deduplication to terabyte scale corporate data.

Figure 13 shows the direct improvement in number of disk seeks provided by NNP grouping over the length of the **fiu** trace, which was 21 days long and included a total of 17,836,701 accesses. As the figure shows, if grouped elements are prefetched into cache, the total number of disk seeks is significantly lower when there is contention for cache space. This indicates that data grouping is worthwhile as long as a rare grouping overhead is less costly than adding more cache to the system. The trace also included

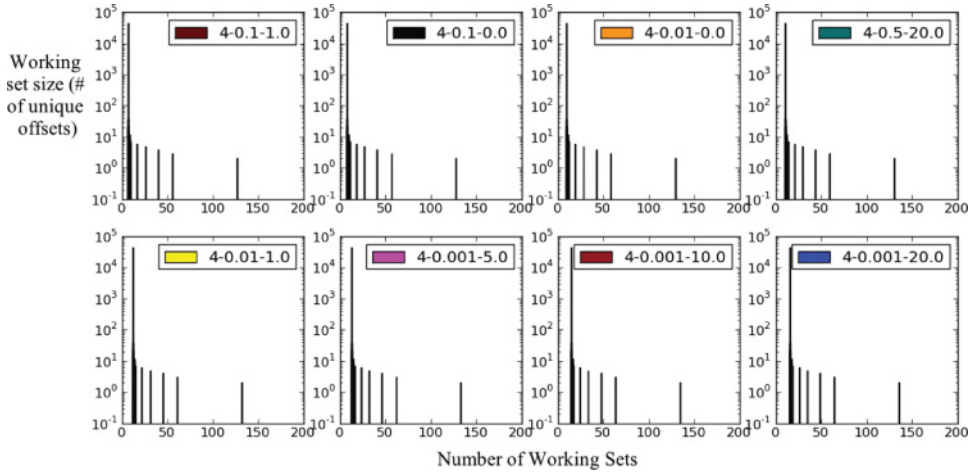


Fig. 12. The bag-of-edges technique run on **msr** data with varying offset and time scaling. Each graph is labeled as number of levels: offset scaling factor, time scaling factor. The resulting group distribution for a given level is very similar regardless of the input parameters.

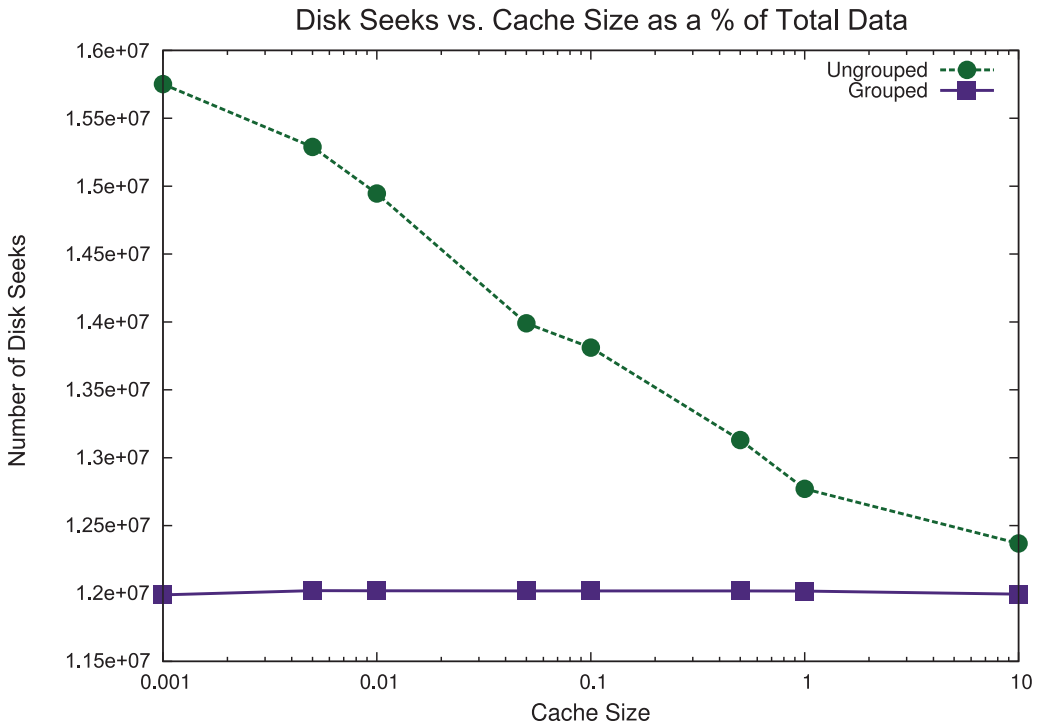
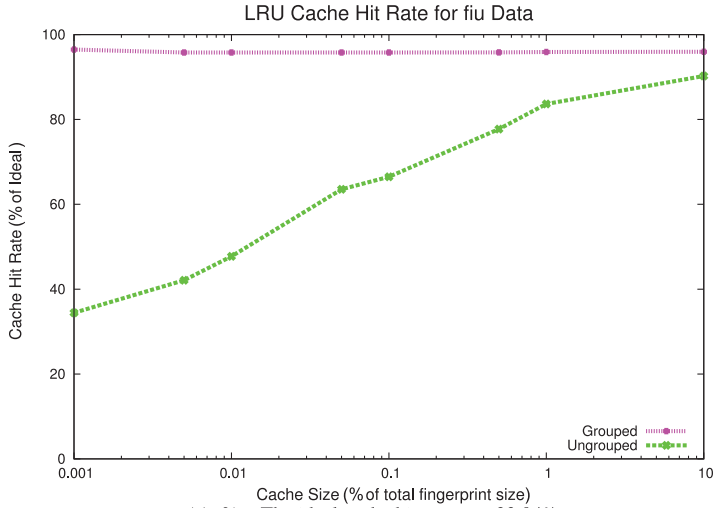
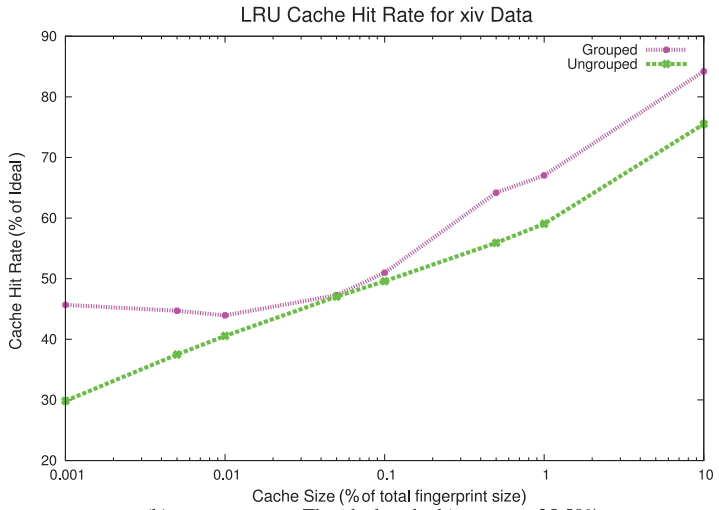


Fig. 13. Disk seeks in grouped versus ungrouped data for the **fiu** trace with an LRU cache. Note that with small cache sizes, grouping has an outsized advantage that decreases as more data is simply left in the cache.



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

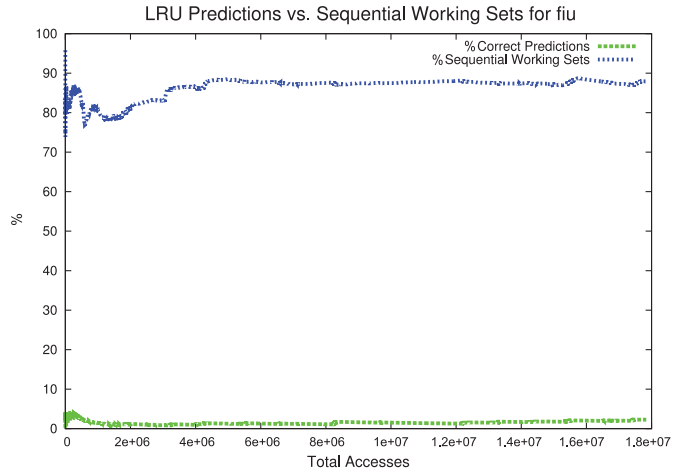


(b) `ent-storage`: The ideal cache hit rate was 35.59%

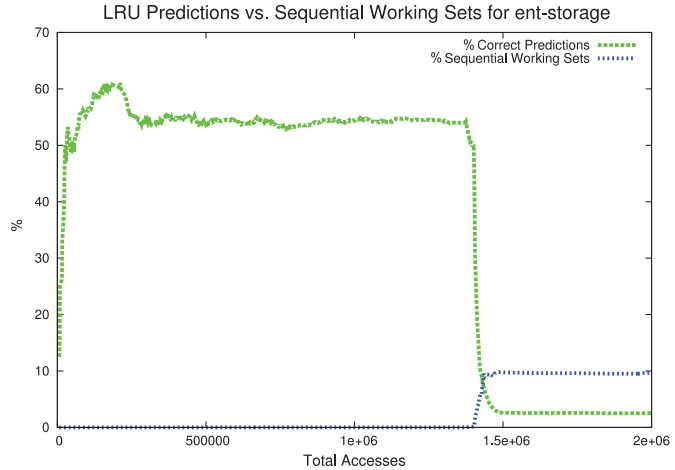
Fig. 14. LRU cache hits across cache sizes. Grouped data does consistently at least as well and often significantly better than ungrouped data. Ideal cache hits are determined by Belady's algorithm [Belady 1966].

fingerprint hashes, and Figure 14 shows the results of NNP grouping used to predictively populate a cache of fingerprints for an in-line deduplication system [Wildani et al. 2013]. Though the groups are small and unstable, grouping was consistently positive and the groups have predictive power (Figure 15).

Table VI shows the actual runtime for NNP totaled across the life of the trace. The overhead depends on how often regrouping was done. For instance, over the entire Washington trace, NNP calculations added 26.473 seconds to the trace runtime. The simulator we use actually runs slower than $O(n)$ because we limit it to 5GB of memory, and it is written using Python list structures. As a result, larger datasets spend a correspondingly large amount of time writing or reading intermediate grouping data from disk. Of course, the predictivity threshold below which regrouping is triggered can be



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



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

Fig. 15. For the `fiu` dataset using LRU, predictive power of groups was unrelated to sequentiality. In the `ent-storage` dataset 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%.

Table VI. Runtimes for NNP in a Python Simulator Across Various Datasets on an Intel Core i7-2600 CPU @ 3.40GHz with 32GB RAM

Name	real	user	sys	Avg. IOPS
ent-storage	0m9.513s	0m9.133s	0m0.340s	75997
fiu	7m4.228s	6m56.818s	0m3.332s	40.49
wash	0m26.473s	0m23.253s	0m0.852s	0.085
msr	0m6.928s	0m6.604s	0m0.292s	6.53

adjusted based on the tolerances of the system. For example, a system could set regrouping to only happen overnight unless the current grouping is actively harmful. The simulator is structured thus for experimental efficiency on a multicore system, but there is no reason that a real implementation needs to obey this limitation. Additionally, even

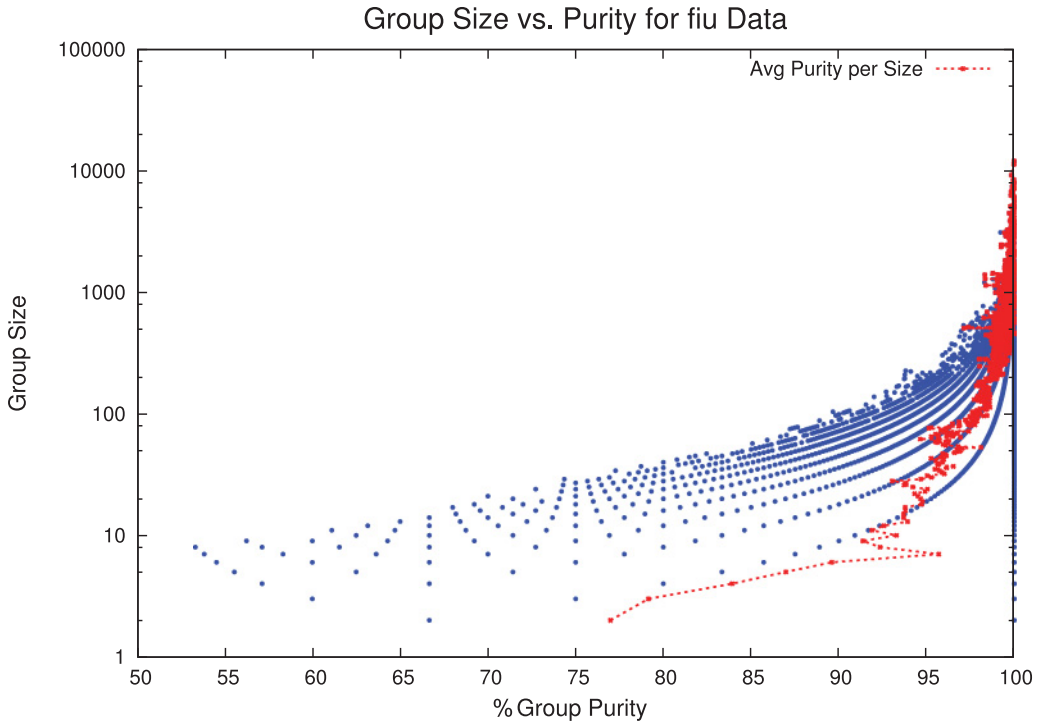


Fig. 16. The **fiu** grouping shows a surprisingly strong positive correlation between group size and purity, where purity is defined as the homogeneity of a group with respect to a third, highly predictive feature such as process ID that was not used in training. This indicates that the filters in place to prevent large, spurious groups are working as intended.

with the inefficiency of our implementation, we see that we can quickly handle windows of tens of thousands of I/Os since that is what the **ent-storage** trace presented.

5.5. Validity

Working sets emerge from how users and applications interact with the data. Consequently, there is no "correct" labeling of accesses to compare our results to. Instead, we focus on self-consistency and stability under parameter variation. As we saw in Figure 12, 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.

The **msr** dataset did not contain any external validation information such as a tertiary metadata feature. We chose to not use cross-validation because any artificial split will impact the groupings; training-testing splits must be temporal only, and they come about automatically since the new data is accessed based on the groups defined from previous data. Thus, we ran NNP on two other datasets (**fiu** and **wash**) that did. We validate statistical groupings for both of these datasets using a third dimension of the input that was not used in calculating the statistical groups. In the `fiu` dataset , we have access to the PID that originated the I/O request. Figure 16 shows the breakdown of groups by size according to the group *purity*, where a pure group is a group where all members share the same originating PID for at least one access. Figure 17 shows

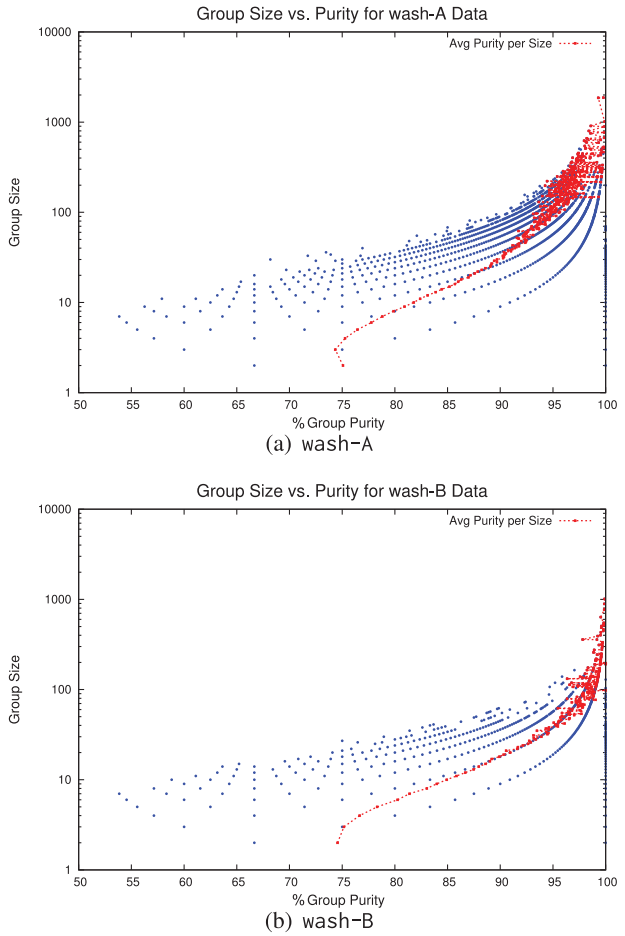


Fig. 17. Both `wash-A` and `wash-B` show a strong correlation between group size and group purity, where purity is defined as the homogeneity of a group with respect to a third, highly predictive feature such as process ID that was not used in training. This indicates that the correlation is resistant to parameter modification.

the same purity calculation where the groups are compared against the somewhat predictive "Record Type" field.

6. SELECTING A GROUPING METHOD

Our grouping techniques produce groupings that are broadly either fast to recalculate or 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 applications based on cache, we needed to maximize the predictive power of groups in a small, fixed size cache, so we further restricted NNP to bias heavily toward 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 colocated data has a high probability of coaccess. We showed instances of this application type when using groups to reduce power consumption [Wildani and Miller 2010] and improve availability [Wildani et al. 2014]. 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 toward 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 toward 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 toward 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.

7. DISCUSSION

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 dataset. It turned out, however, that in every classification scheme we saw no chains of more than three overlapping groups. 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 overlapping components since the accesses are uniformly distributed.

Another encouraging result of our study was the consistency of groupings in the data despite the sparsity of the traces. This indicates that it is worthwhile to look for groupings in similar multiuser and multiapplication workloads even if the only data available is block offsets and timestamps. 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.

For this research work, we were not concerned with the speed of computations as long as the algorithms were efficient. While testing algorithms, we wrote all code in Python and did not optimize for speed. Under this constraint, the localized distance techniques such as NP ran nearly instantaneously, while the global techniques, particularly the graph techniques, took between 20 and 35 minutes per run on a server with an 8-core 2.4GHz Intel®Xeon processor and 24GB of RAM. We believe that much of this speed can be regained by tighter code. Also, online implementations will handle less data per timestamp than static test cases.

We realize the assumption that implicitly defined UIDs such as block offset do not uniquely identify a piece of data 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.

One surprising feature of the **msr** dataset was that it had a long list of consecutive writes to the same block. We believe that these writes are the result of overwrite activity in a log or a disk-based cache. These types of points are frequently present but filtered

in other block I/O traces [Amer and Long 2002]. We intentionally include these points in our classifications to verify that they trivially classify into their own working set. Our *k*-NN and bag-of-edges methods can work around the noise of the spike to produce realistic working set groups given reasonable parameters.

8. CONCLUSIONS

We have presented two distance metrics and three partitioning algorithms for separating a stream of I/O data into working sets. We have found that the working sets discovered by our method are stable under perturbation, implying that they have a high-level basis for existing.

Overall, NNP outperformed every other method in terms of scalability even though it, like NP, lacks stability. At the other extreme, bag-of-edges is a stable method for situations where runtime is less of a concern. We also present an evaluation of how to choose which grouping technique is best for which workload. Our methods are broadly applicable across workloads, and we have presented an analysis of how different workloads should respond to different partitioning methods. Unlike previous work in the field, we perform analysis using only data that can be collected without impacting the performance of the system. Our methods are also designed to separate working sets that are interleaved within the I/O stream. Finally, our methods are designed for use on disks instead of cache, changing the design goals from "likely to be accessed next" to "likely to be accessed together."

A consistent, easily calculable grouping that is not tied to a specific workload opens up two main avenues of work that is essential for the next stage of exascale system development. First, we will be able to characterize workloads based on how they are separated and how separable they are. Knowing a workload is likely separable allows us to move onto the next step, which is dynamically rearranging data across a large storage system according to the working set classification of the data. Being able to rearrange data to minimize spin-ups will be essential to keeping down power cost and increasing the long-term reliability of these increasingly vital storage systems.

8.1. Future Work

Our current project is to use a protocol analyzer to collect block I/O data from a mixed-use, multidisk educational storage system to provide a direct comparison and validity numbers to extend this work. With this data stream, we hope to implement working set detection in real time, as well as track potential power savings and reliability gains from grouping the data together according to the assigned working set. We are analyzing our groupings using a variety of techniques that will be meaningful once we have additional datasets to compare statistics with. This includes calculating the direct overlap of elements between different groupings of working sets, calculating mutual entropy between different groupings, and calculating a discrete Rand index value across groupings [Rand 1971]. In the absence of other data, the numbers tell us little more than our graphs do. Calculating these indices for two workloads would be a good first step toward characterizing workloads based on their separability into working sets. The final determinant of group validity will be the improvement in power consumption and system usability that results in rearranging data in separable workloads to place working sets together on disk.

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 [Wildani and Miller 2010]. We are also interested in refining the graph covering algorithm to accept groups that are only partially connected instead of requiring complete cliques by implementing techniques from community detection [Lancichinetti and Fortunato 2009].

REFERENCES

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

A. Amer and D. D. E. Long. 2002. Aggregating caches: A mechanism for implicit file prefetching. In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 293–301.

A. Amer, D. D. E. Long, J. F. Paris, and R. C. Burns. 2002. File access prediction with adjustable accuracy. In *IEEE International Conference on Performance, Computing and Communications (IPCCC)*. IEEE Computer Society, 131–140.

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

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

M. Barbaro and T. Zeller Jr. 2006. A face is exposed for aol searcher no. 4417749. (August 2006).

L. A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal* 5, 2 (1966), 78–101.

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

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

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

T. H. Cormen, C. E. Leiserson, and R. L. Rivest. 1990. *Algorithms*. MIT Press, Cambridge, MA.

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

S. Doraimani and A. Iamnitchi. 2008. File grouping for scientific data management: Lessons from experimenting with real traces. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*. ACM, 153–164.

R. O. Duda, P. E. Hart, and D. G. Stork. 2001. *Pattern Classification*. Vol. 2. Citeseer.

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

Bert Dufrasne, Roger Eriksson, Lisa Martinez, and Wenzel Kalabza. 2012. *IBM XIV Storage System Gen3 Architecture, Implementation, and Usage*. IBM, International Technical Support Organization. 426 pages.

P. Jaccard. 1901. Distribution de la flore alpine dans le bassin des Dranses et dans quelques régions voisines. *Bulletin del la Société Vaudoise des Sciences Naturelles* 37 (1901), 241–272.

S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. 2005. DULO: An effective buffer cache management scheme to exploit both temporal and spatial locality. In *USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, 8.

R. Koller and R. Rangaswami. 2010. I/O deduplication: Utilizing content similarity to improve I/O performance. *ACM Transactions on Storage (TOS)* 6, 3 (2010), 1–26.

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

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

A. Lancichinetti and S. Fortunato. 2009. Community detection algorithms: A comparative analysis. *Physical Review E* 80, 5 (2009), 056117.

W. Li. 2008. *An Efficient Query System for High-Dimensional Spatio-Temporal Data*. Ph.D. Dissertation. University of Massachusetts Lowell.

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

S.-w. Lo, B.-H. Chen, Y.-W. Chen, T.-C. Shen, and Y.-C. Lin. 2014. ICAP, a new flash wear-leveling algorithm inspired by locality. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 1478–1483.

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

A. E. Magurran. 2004. Measuring biological diversity. In *African Journal of Aquatic Science* 29, 2, 285–286.

J. Metz. 2012. Working document of the new technologies file system (NTFS). 0.0.3 (2012).

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

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

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

W. M. Rand. 1971. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association* 66 (1971), 846–850.

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

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

F. Schmuck and R. Haskin. 2002. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST'02)*. USENIX, 231–244. http://www.ssrc.ucsc.edu/PaperArchive/schmuck-fast02.pdf.

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

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

N. Slonim, G. Singh Atwal, G. Tkacik, and W. Bialek. 2005. Information-based clustering. *Proceedings of the National Academy of Science* 1021 (Dec. 2005), 18297–18302.

T. Sørenson. 1948. A method of establishing groups of equal amplitude in plant sociology based oil similarity of species content. *Biologiske Skrifter/Kongelige Danske Videnskabernes Selskab* (1948), 1–34.

C. Staelin and H. Garcia-Molina. 1990. Clustering active disk data to improve disk performance. Princeton, NJ, Tech. Rep. CS–TR–298–90 (1990).

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

J. Wang and Y. Hu. 2001. PROFS-performance-oriented data reorganization for log-structured file system on multi-zone disks. In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. Published by the IEEE Computer Society, 0285.

A. Wildani and E. L. Miller. 2010. Semantic data placement for power management in archival storage. In *2010 5th Petascale Data Storage Workshop (PDSW'10)*. IEEE, 1–5.

A. Wildani, E. L. Miller, and O. Rodeh. 2013. HANDS: A heuristically arranged non-backup in-line deduplication system. In *2013 IEEE 29th International Conference on Data Engineering (ICDE'13)*. IEEE, 446–457.

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

A. Wildani, E. L. Miller, I. Adams, and D. D. E. Long. 2014. PERSES: Data layout for low impact failures. In *22th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'14)*.

G. Wu and X. He. 2012. Delta-FTL: Improving SSD lifetime via exploiting content locality. In *Proceedings of the 7th ACM European Conference on Computer Systems*. ACM, 253–266.

N. J. Yadwadkar, C. Bhattacharyya, K. Gopinath, T. Niranjan, and S. Susarla. 2010. Discovery of application workloads from network file traces. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*. USENIX Association, 14.

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

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

Received April 2014; revised December 2014; accepted February 2015