UNIVERSITY OF CALIFORNIA
SANTA CRUZ

DEDUPLICATION FOR LARGE SCALE BACKUP AND ARCHIVAL STORAGE

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

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Deepavali Bhagwat

September 2010

The Dissertation of
Deepavali Bhagwat is approved:

Professor Darrell D. E. Long, Chair

Kave Eshghi

Professor Ethan Miller

Mary Baker

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Deduplication for Large Scale Backup and Archival Storage

by

Deepavali Bhagwat

The focus of this dissertation is to provide scalable solutions for problems unique to chunk-based deduplication. Chunk-based deduplication is used in backup and archival storage systems to reduce storage space requirements. We show how to conduct similarity-based searches over large repositories, and how to scale out these searches as the repository grows; how to deduplicate low-locality file-based workloads, and how to scale out deduplication via parallelization, data and index organization; how to build a unified deduplication solution that can adapt to tape-based and file-based workloads; and, how to introduce strategic redundancies in deduplicated data to improve the overall robustness of the system.

Our scalable similarity-based search solution finds for an object, highly similar objects from within a large store by examining only a small subset of its features. We show how to partition the feature index to scale out the search, and how to select a small subset of the partitions (less than 3%), independent of object size, based on the content of query object alone to conduct distributed similarity-based searches.

We show how to deduplicate low-locality file-based workloads using Extreme Binning. Extreme Binning uses file similarity to find duplicates accurately and makes only one disk access for chunk lookup per file to yield reasonable throughput. Multi-node backup systems built with Extreme Binning scale gracefully with the data size. Each backup node is

autonomous—there is no dependency between nodes, making house keeping tasks robust and low overhead.

We build a 'unified deduplication' solution that can adapt and deduplicate a variety of workloads. We have workloads consisting of large byte streams with high-locality, and workloads made up of files of varying sizes without any locality between them. There are separate deduplication solutions for each kind of workload, but so far no unified solution that works well for all. Our unified deduplication solution simplifies administration—organizations do not have to deploy dedicated solutions for each kind of workload—and, it yields better storage space savings than dedicated solutions because it deduplicates across workloads.

Deduplication reduces storage space requirements by allowing common chunks to be shared between similar objects. This reduces the reliability of the storage system because the loss of a few shared chunks can lead to the loss of many objects. We show how to eliminate this problem by choosing for each chunk a replication level that is a function of the amount of data that would be lost if that chunk were lost. Experiments show that this technique can achieve significantly higher robustness than a conventional approach combining data mirroring and Lempel-Ziv compression while requiring about half the storage space.

To my husband, who believed in me when I did not.

Acknowledgments

I am grateful to have found Darrell Long, my advisor, whose constant support, understanding and patience made this dissertation possible. His support during both my pregnancies and the most difficult period when I was learning to be a new mom is the only reason I stayed in the PhD program. Darrell has not only been my advisor, but also a friend.

It has been an honor and privilege to work with and learn from Kave Eshghi. His continued kindness, patience and encouragement helped me build my thesis. I consider myself lucky to be able to continue to collaborate with him at Hewlett Packard.

I would like to thank Ethan Miller and Mary Baker for being on my dissertation committee and approving my advancement to candidacy. Thomas Schwarz has been a friend, advisor and counselor. His strength helped me navigate through the most stressful periods of my life.

This milestone would not have been possible without the friendship, help and support of the following people: Jaap Suermondt encouraged me to continue my internship at Hewlett Packard Laboratories. Scott Brandt introduced me to Darrell. I am grateful for his trust. My friends at the Storage Systems Research Center—Kevin Greenan, Mark Storer, Andrew Leung, Chris Xin, Kristal Pollack—our discussions and camaraderie made this journey enjoyable. Pankaj Mehra encouraged me to join Hewlett Packard Laboratories. Vinay Deolalikar's friendship made me feel at home there. Doing research with Mark Lillibridge taught me a lot. Neoklis Polyzotis guided me through my masters project.

The ARCS (Achievement Rewards for College Scientists) Foundation's scholarship for two years lightened the psychological burden of being a PhD student for too long. The

environment at Hewlett Packard Laboratories helped fuel my research. Bruce Baumgart of the Internet Archive, Mike Blaesser and Bob Smith of the Santa Cruz Sentinel gave me access to their data. Lawrence You helped me understand the Deep Store prototype. The National Science Foundation (Grant CCR-0310888) and all of the Storage System Research Center's industrial sponsors supported my work.

My parents inculcated in me the value of a good education. Their help taking care of my infant son while I was trying to get research done was invaluable.

My son Shashi's care givers made a huge effort to get to know him and to make him feel at home at the Ames Child Care Center. The knowledge that my son was happy and thriving while he was away from his parents allowed me the peace of mind needed to continue with my research.

Shashi's arrival changed my life in ways I had never expected. I learned that it was not always possible to be prepared for every eventuality in life and that was okay. Being a mom is the most rewarding aspect of my life.

This milestone has been my husband's dream. After his PhD, he missed grad school so much that he wanted to relive that experience through his wife. His unwavering belief in me, his strength, companionship, and love made it possible to find the strength I needed to persist and survive this journey. Benjamin Franklin once wrote, "Be always at war with your vices, at peace with your neighbors, and let each new year find you a better man". I have been married to Mahendra Bhagwat for nine years and every year, without exception, I have found a better man in him.

Chapter 1

Introduction

This dissertation presents scalable solutions for problems unique to chunk-based deduplication. We show how to conduct similarity-based searches over large repositories, and how to scale out these searches as the repository grows; how to deduplicate low-locality file-based workloads, and how to scale out deduplication via parallelization, data and index organization; how to build a unified deduplication solution that can adapt to tape-based and file-based workloads; and, how to introduce strategic redundancies in deduplicated data to improve the overall robustness of the system.

Digital data continues to grow at an unprecedented rate. The amount of digital information created in 2007 was 281 exabytes; by 2011, it is expected to be 10 times larger [38]. 35% of this data originates in enterprises and consists of unstructured content, such as office documents, web pages, digital images, audio and video files, and electronic mail. Enterprises retain data for the purposes of corporate governance, regulatory compliance [1, 2], litigation support, and data management. As data grows, so does the need to find cost-effective storage

solutions. *Deduplication*, a technique for eliminating duplicate data and reducing storage space requirements, is gaining prominence. Backup data is particularly well suited for deduplication. Storage space requirements can be reduced by a factor of 10 to 20 or more when backup data is deduplicated [13].

Chunk-based deduplication [37, 57, 66], a popular deduplication technique, first divides input data streams into fixed or variable-length chunks. Typical chunk sizes are 4 or 8 kB. A cryptographic hash or *chunk ID* of each chunk, computed using techniques such as SHA-1 [59], is used to determine if that chunk has been backed up before. Chunks with the same chunk ID are assumed identical [41, 14]. New chunks are stored and references are updated for duplicate chunks. Chunk-based deduplication is very effective for backup workloads, which tend to be files that evolve slowly, mainly through small changes, additions, and deletions [83].

This dissertation focuses on inline chunk-based deduplication. *Inline* deduplication is deduplication where data is deduplicated before it is written to disk as opposed to post-process deduplication where backup data is first written to a temporary staging area and then deduplicated offline. One advantage of inline deduplication is that extra disk space is not required to hold and protect data yet to be backed up. This is in contrast to out-of-band or offline deduplication, where data is first written to temporary storage and then deduplicated offline.

One of the problems concerning scalable inline deduplication has been aptly labelled the 'disk bottleneck problem' [87]. To facilitate fast chunk ID lookup, a single index containing the chunk IDs of all the backed up chunks must be maintained. However, as the backed up data grows, the index overflows the amount of RAM available and must be paged to disk. The index cannot be cached effectively, and as data grows, it is not uncommon for nearly every

index access to require a random disk access. This disk bottleneck severely limits deduplication throughput. To solve this problem, this thesis presents two solutions: Content-based Document Routing: a method for conducting scalable similarity-based searches over large document repositories [11], and, Extreme Binning: a method for scalable chunk-based file backup [10].

1.1 Scalable Similarity-based Search

To deduplicate *efficiently and at scale*, it is essential that the process of *finding similar objects* within large object repositories/stores be efficient. Similar objects are objects that share content. This shared or duplicate content is what we want to identify so that it can be eliminated. By eliminating duplicate content, storage space can be conserved. While finding similar objects can in principle be achieved by a pairwise comparison of the one object with each of the other objects in the repository, this process quickly becomes very inefficient as repositories grow. To scale gracefully with the repository size, we propose a new scalable similarity-based search method. The goal is to find objects that are *highly* similar to each other. We are not interested in objects that incidentally share some small content. Focussing our efforts only on the highly similar objects, allows gracefull scale out—we can continue to find shared content quickly even as repositories grow. Further, highly similar objects are of the most interest to us because by eliminating the large chunks of overlapping content between them we get the most 'bang for our buck' in terms of storage space savings.

To find similar objects within a repository, we use the following framework : from every object, a set of features are extracted, such that if two documents are similar, their sets of features overlap strongly, and if they are dissimilar, their sets of features do not overlap. Thus,

the problem of object similarity is reduced to one of set similarity. The similarity-based search solution uses this set similarity to find similar documents with high precision.

For fast query and retrieval features are maintained in an index. To scale out the search, we show how to partition this search index into smaller indices and how to conduct a distributed search. This is done in such a way so that the number of partitions that need to be queried per object is very small and moreover, is independent of the object size. This number, also called the 'routing factor' is a tunable parameter which can be used to control how exhaustive the searches need to be. The decision as to which partitions should be queried is based solely on the content of the object and not on the contents of the partitions. This makes our approach stateless. The scalable similarity-based search and routing algorithm are discussed in Chapter 2.

1.2 Scalable, Parallel Deduplication for Chunk-based Low-locality File Backup

Deduplication is almost ubiquitous in backup systems. Every night, backup agents crawl through enterprise and user data to package it into large files using utilities such as 'tar'. These large files or byte streams are then sent to the backup systems where they get deduplicated before being written to backup media. Such a workload made up of large byte streams is what we call the 'traditional' backup workload. It is generated by Virtual Tape Library agents and then streamed across the network for backup. Traditional backup workloads contain high locality.

Locality is the tendency of groups of chunks to reoccur together. For example, if the last time chunk A occured, it was surrounded by chunks B, C, and D, then the next time A occurs (even in a different backup) it is likely that B, C, or D will be nearby. Deduplication solutions, such as Sparse Indexing [49] and Locality Sensitive Caching [87], designed for the traditional workload depend on this locality to perform well. They use locality to reduce disk accesses and improve throughput.

Now consider clients such as NAS file shares and electronic mail servers. These clients send individual files of varying sizes, rather than large byte streams, for backup. No locality can be assumed across files arriving within a window of time. When Sparse Indexing or Locality Sensitive Caching techniques are presented with such a 'non-traditional' file-based workload, their performance deteriorates—their deduplication quality and throughput suffers.

Our next contribution is Extreme Binning: a scalable deduplication technique for non-traditional backup workloads. Extreme Binning depends on *file similarity* to guarantee throughput and deduplication quality. It makes only one disk access for chunk lookup *per file*, which gives reasonable throughput. It uses a similarity-based search method to find duplicates with high accuracy. In Chapter 3, we also show why and how Extreme Binning can be easily scaled out to accomodate the growth of data. We show how to build a multi-node backup system where files can be deduplicated in parallel and there is no dependency with respect to data or indices across nodes. Each file is allocated to only one node using a stateless routing algorithm [43, 56] for deduplication *and* storage, allowing for maximum parallelization. Each backup node is, thus, autonomous, making data management and house keeping tasks robust and low overhead.

1.3 Unified Deduplication

Unified Deduplication is a term we use to describe a deduplication engine that can deduplicate *across* a variety of workloads—VTL tape images, electronic mail messages, files or folders sent by NAS boxes and backup agents installed on PCs. It should be able to adapt to traditional as well as non-traditional workloads. Such a unified deduplication engine is useful because it simplifies management—a dedicated system does not need to be deployed and administered to service each category of clients. It also yields better storage space savings over dedicated systems because it can deduplicate *across* workloads. For example, consider an electronic mail attachment saved on the desktop of an employee. This attachment will occur both as a part of the exchange server's workload and the weekly/nightly backup generated by a VTL agent. A unified deduplication solution detects such duplicates whereas dedicated systems cannot. These advantages, of ease of manageability/administration and storage space savings, make unified deduplication a desirable solution. In Chapter 4 we show how to design such a unified deduplication system.

To do this, we first study the state-of-the-art solutions with the objective of assessing whether they can be adapted to handle the variations in workloads. There is Sparse Indexing for large byte streams with high locality—essentially the traditional nightly and weekly backup workload; and there is Extreme Binning for the non-traditional, file-based, low-locality workload. Locality Sensitive Caching is another solution developed for traditional backup workloads. However, we do not have access to its actual implementation. Both, Sparse Indexing and Extreme Binning, use a sampling approach to reduce RAM usage. Both amortize the cost of disk accesses by deduplicating groups of chunks at a time.

To assess these approaches, we presented each solution with a workload that it is not designed for. We presented Sparse Indexing with a file-based, low-locality workload and Extreme Binning with a traditional workload. For the traditional workload, the byte stream was segmented into objects of 10 MB. So, as far as Extreme Binning was concerned, it was receiving 10 MB files, one file at a time. Our observation was that Extreme Binning is not well suited to handle the traditional workload. It only extracts one sample per object which is not enough to deduplicate objects the size of 10 MB. We can perhaps make these objects smaller and then feed them to Extreme Binning to improve deduplication, however, this would be at the cost of reduced deduplication throughput. A high throughput is essential when backing up on a nightly basis due to hard deadlines and Sparse Indexing achieves that.

When presented with a non-traditional workload, Sparse Indexing did not do as good a job as Extreme Binning either. This was primarily because the files in the workload were some times too small to yield any samples. However, we found that by changing the sampling method and by making a few more modifications, we could improve Sparse Indexing's performance. Hence, we extended Sparse Indexing to build a unified deduplication solution. We not only tested it with traditional and non-traditional workloads, but with combination workloads as well. Combination workloads are those that contain a mixture of objects varying in size as well as locality. We evaluated this unified deduplication solution on the basis of its deduplication quality and RAM usage.

1.4 Strategic Redundancies for Reliability

Backup and archival systems, in addition to conserving storage space, must also ensure that data is preserved over long time periods. Data must be protected so that if required, at a later date, it be accessible and reproduced faithfully. Deduplication techniques, while they save storage space, also have the potential to reduce reliability. Deduplication removes redundant content across objects. Only references are updated for duplicate chunks. This means that dependencies are introduced between objects through such shared chunks. If such a shared chunk is lost all the objects that share this chunk will not be reconstructible. If a highly shared or popular chunk is lost, then a disproportionately large number of objects and data may become inaccessible. In Chapter 5 we show how to solve this problem by re-introducing some duplicate content.

The key idea is that though all chunks must be protected against loss due to hardware failures or software errors, some chunks are much more important than the others and thus, must be protected more aggressively. The measure of a chunk's importance is the number, or, the cumulative size of the objects that together share it amongst themselves. The more 'important' chunks, then, need to be protected at a higher level than the others to maintain good overall reliability. To this end, we propose a solution [12] that strategically replicates critical chunks more aggressively than less popular chunks.

By re-introducing redundancies we are in some ways reversing the effects of deduplication because these redundant chunks consume extra storage space. However, we believe that our approach of introducing *strategic* redundancies is a way of re-investing a small fraction of the saved storage space in the interests of improved reliability. Indeed, our results show

that we achieve even better data reliability than mirrored (degree of mirroring = 2) Lempel-Ziv (LZ) compressed [88] objects, while still using about half of the storage space of mirrored LZ-compressed objects and with replication/mirroring as the means to introduce redundancies.

1.5 Summary

To build scalable chunk-based deduplication systems this thesis proposes the following solutions:

1. A similarity-based scalable search method for finding similar documents in large corpuses. By choosing only a few features of a document, the similarity-based search finds highly similar documents fast. The document routing algorithm allows for scalability—the search index can be partitioned into smaller sizes and searches can be parallelized.

2. A scalable, parallel deduplication method for chunk-based file backup where locality cannot not be assumed. Extreme Binning requires only one disk access per file and can be parallelized without impacting deduplication ratios. Bin independence makes scale out, housekeeping and integrity check operations clean and simple.

3. A study of existing technologies to assess their potential use in building a unified deduplication engine. We show how to adapt the Sparse Indexing technology using variations of the min-hash sampling algorithm to build a deduplication system that can adapt to varied workloads.

4. A method for improving the robustness of chunk-based deduplication systems by adding strategic redundancies. The results show that by re-investing a small amount of conserved

storage space it is possible to improve robustness while still being more space efficient that mirrored Lempel-Ziv compression.

The common thread among all the solutions in this dissertation, is that each involves making a trade-off. The similarity-based scalable search makes a trade-off between efficiency, scalability and precision—only highly similar objects are found with the less or vaguely similar objects disregarded. Extreme Binning makes a trade-off between throughput, RAM usage and deduplication ratio. To ensure throughput, only one disk access is allowed per file. To reduce RAM utilization, for every object, only one of its chunk IDs is kept in RAM. In exchange for these desirable properties, a few of duplicates are allowed. Unified Deduplication provides a single point solution for varied workloads and allows for global deduplication—across workloads generated by varied sources. In exchange for this ease of administration, some duplicates may be allowed—some workloads may not be deduplicated as well as they would have been by dedicated deduplication solutions. However, our results show that Unified Deduplication, due to its capability to deduplicate across workloads, achieves better deduplication than dedicated solutions. Selective chunk replication makes a trade-off between storage space consumption and reliability. Even though extra storage space is consumed reliability is improved.

The rest of this thesis is organized as follows: Chapter 2 discusses the scalable similarity-based algorithm. Extreme Binning is described in Chapter 3 while Unified Deduplication is described in Chapter 4. Chapter 5 describes how selective redundancies help improve reliability. Finally, Chapter 6 concludes with contributions and future directions.

Chapter 2

Scalable Similarity-based Searches over Large Document Repositories

2.1 Introduction

Finding textually similar files in large document repositories is a well researched problem, motivated by many practical applications. One motivation is the need to identify near duplicate documents within the repository, to eliminate redundant or outdated files and improve user experience [37]. Archival systems [66, 84] need to identify content overlap between files to save storage space by using techniques such as delta-compression [4, 29]. Other applications arise in information management: similarity based retrieval can be used to find all versions of a given document, e.g. for compliance, security, or plagiarism detection purposes. Notice that here we are concerned with textual document similarity, where two documents are deemed to be similar if they share significant stretches of text. This is in contrast to natural language based approaches where the linguistic structure of the document is taken into account.

The operation that is the object of our study is *similarity based retrieval*. Here, a *query document* Q is presented to the system. The aim is to find all the documents $D_1, D_2, \ldots, D_n$ in the repository that are similar to Q, the most similar documents being presented first.

While finding textually similar documents can in principle be achieved by a pairwise comparison of the query document with each one of the documents in the repository using a program such as Unix `diff`, this is clearly very inefficient. To solve this problem, the following framework is commonly used: from every document, a set of features are extracted, such that if two documents are similar, their sets of features overlap strongly, and if they are dissimilar, their sets of features do not overlap. Thus, the problem of document similarity is reduced to one of set similarity. Then, an inverted index [72, 73] is created mapping features to documents. Upon the presentation of the query document Q, its features are extracted, and used to query the inverted index. The result is a set of documents that share some features with Q; these are then ranked with the document sharing most features coming first. Various authors have developed techniques for extracting features from a document: Manber [52], Broder *et al.* [18, 19], Pearce *et al.* [63], Kulkarni *et al.* [47] and Forman *et al.* [37]. All these approaches generate very specific features: when two documents share even a single feature, they share a relatively large stretch of text (tens of characters). As a result, when the inverted index is consulted, relatively few documents are returned. This is in contrast with techniques such as bag of words analysis, where most documents are expected to share at least a few words.

A single feature index becomes a bottleneck as the size of the repository gets very large, and the index needs to simultaneously handle a large number of updates and queries. Such a situation is typical of document management systems for large enterprises, as well as archival

systems dealing with large, continuous streams of documents. One solution is to partition the feature index into a number of sub-indices, placing each partition on a different server, so that the servers can be updated and queried in parallel. The partitioning scheme used must be such that only a small fraction of the partitions need to be accessed for each update and query. Here, the most obvious schemes, such as using a Distributed Hash Table (DHT) [68, 71, 76, 86] to store the $\langle feature, Document \rangle$ pairs fail. In this scheme, the partitioning of the index is based on the hash of the individual features. For example, given 2^k servers, each hosting a hash table, and the pair $\langle feature, Document \rangle$ that needs to be added to the index, the first k bits of the hash of the feature are used for identifying the server that this particular pair needs to be added to. The problem is that there is no locality of reference for the individual feature hashes: each one of the document features is routed independently, most likely to a different server. As a result, a large number of servers will need to be accessed for each document update and query.

We present an alternative index partitioning and document routing scheme; one that does not route each individual feature of every document independently, but rather routes all the features in a document together. The outline of our scheme is as follows:

- At *ingestion time*, i.e. when a document is being added to the repository, the document's features are extracted using a feature extraction algorithm. Based on these features, a fraction of the index partitions are chosen, and the document is *routed* to these partitions, i.e. the document's features are sent to these partitions and added to the index there. We call the algorithm by which the partitions are chosen the *document routing algorithm*.

- At *query time*, the same feature extraction and document routing algorithms are used for choosing the partitions to query. The chosen partitions are queried with the document's features, and the results are merged.

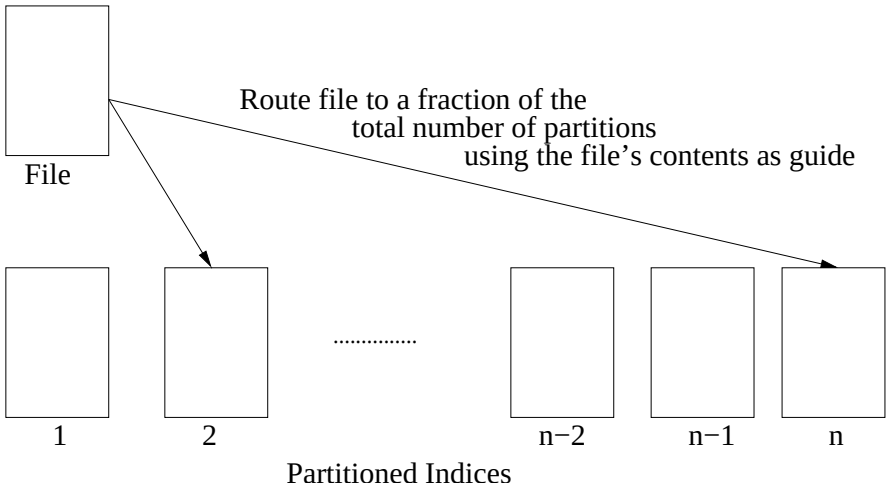

Figure 2.1: Our solution for document routing

This situation has been depicted in Figure 2.1. Notice that the choice of which servers to contact, both at ingestion and query time, is entirely based on the contents of the document; at no time do we have to have an interaction with the partitions to determine which one should be chosen. This sets us apart from approaches which apply a clustering algorithm [45, 62, 74] to all the documents in the repository; these approaches need to use knowledge of the existing clusters to route the new document. Our approach, by contrast, is incremental and stateless: only the contents of the document are used to decide which partitions it should be routed to. As a result, we have a very lightweight, client based routing capability.

2.2 Background

As stated above, our work assumes the existence of a mechanism to extract features from documents, such that document similarity is reduced to set similarity. We use the Jaccard index as a measure of set similarity. Let H be an algorithm for extracting features from documents, where $H(f)$ stands for the set of features extracted by H from document f. Then, the *similarity measure* of two documents f_1 and f_2 according to Jaccard index is

$$\frac{|H(f_1) \cap H(f_2)|}{|H(f_1) \cup H(f_2)|}$$

There are a number of feature extraction algorithms in the literature that satisfy the requirements above. Shingling [17] is a technique developed by Broder for near duplicate detection in web pages. Manber [52], Brin *et al.* [16] and Forman *et al.* [37] have also developed feature extraction methods for similarity detection in large file repositories.

For the experiments, we used a modified version of the chunk based feature extractor described by Forman *et al.* [37]. This algorithm is described below.

2.2.1 Chunk Based Feature Extraction

Content-based chunking, as introduced in [57], is a way of breaking a file into a sequence of chunks so that chunk boundaries are determined by the local contents of the file. The Basic Sliding Window Algorithm is the prototypical content-based chunking algorithm. This algorithm is as follows: an integer, A, is chosen as the desired average chunk size. A fixed width sliding window is moved across the file, and at every position k, the fingerprint, F_k, of the contents of this window is computed. This fingerprint is calculated using a technique known

as Rabin's fingerprinting by random polynomials [67]. The position k is deemed to be a chunk boundary if $F_k \bmod A = 0$. We actually use the TTTD chunking algorithm [34], a variant of the basic algorithm that works better [37].

The rationale for using content-based chunking for similarity detection is that if two files share a stretch of content larger than the average chunk size, it is likely that they will share at least one chunk. This is in contrast to using fixed size chunks, where inserting a single byte at the beginning would change every chunk due to boundary shifting.

We use the characteristic fingerprints of chunks (see below) as the features of the file. Again, the intuition is that if two files are similar, they share a large number of chunks, and thus their feature sets overlap strongly; if they are dissimilar, they will not share any chunks, and thus their feature sets will be disjoint.

Here is the feature extraction algorithm in more detail. This algorithm uses a hash function, h, which is an approximation of a min-wise independent permutation (see Section 2.3.2 below). There are three steps in our feature extraction algorithm:

1. The given file is first parsed by a format specific parser. We handle a range of file formats, including PDF, HTML, Microsoft Word and text. The output of the parser is the text in the document.

2. The document text is divided into chunks using the TTTD chunking algorithm [34]. The average chunk size chosen for these experiments was 100 bytes.

3. For each chunk, a *characteristic fingerprint* is computed, as follows: let $\{s_1, s_2, \ldots s_n\}$ be the overlapping q-grams in the chunk, i.e. the set of all subsequences of length q in

the chunk. Then the characteristic fingerprint of the chunk is the minimum element in the set $\{h(s_1), h(s_2), \ldots h(s_n)\}$, where h is the hash function described above. For the experiments we chose q to be 20.

To summarize, the features of the document are the characteristic fingerprints of the chunks of the document. This algorithm has been demonstrated to produce good features for document similarity. We will not discuss its properties further here, since it is not the subject of this paper.

2.2.2 The Structure of Feature Indices

Here, we describe the structure of the feature indices, be they a monolithic feature index for all the documents in the repository, or one of the indices corresponding to a partition of the bigger index.

Figure 2.2 depicts one of the possible designs of a feature index. The index key is the feature itself. Each feature points to the list of files that it occurs in. This design is analogous to that of an inverted keyword index [72, 73] used commonly in Information Retrieval Systems. This index contains lookup information for every file that has been routed to it at ingestion time.

2.2.3 Building the Feature Indices

When a new file, f_n, needs to be added to the repository an entry for each feature in $H(f_n)$ must be added to the feature index. For every feature in $H(f_n)$, if an entry already exists in the feature index, then the detail for that entry is appended with f_n. If no entry is found

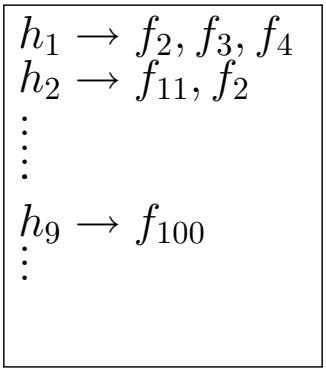

Figure 2.2: Feature Index

then a new entry for that feature is inserted. If f_n is routed to multiple partitions this process is repeated at every one of the destination partitions.

2.2.4 Querying Feature Indices

When a feature index needs to be accessed to find files in the repository similar to a query file, f_q, then the index is queried using each feature in $H(f_q)$. The set of files similar to f_q is the set

$$\bigcup_{0 \leq i < |H(f_q)|} (I(h_i))$$

where $I(h_i)$ is the set of all the files that h_i points to in the feature index I. Each file in the result set is ranked based on its Jaccard similarity index with respect to f_q.

If multiple partitions need to be queried for f_q then the above querying process is carried out for every partition. The results obtained from each partition are collated such that

the set of all the files similar to f_q is given by

$$\bigcup_{i \in R} \left(\bigcup_{0 \leq j < |H(f_q)|} (I_i(h_j)) \right)$$

where the set R is the set of all the partition numbers that were queried for f_q.

2.3 Partitioning the feature index

As mentioned before, our main interest is to partition the index I into a number of sub-indices $I_1, I_2, \ldots, I_K$ while preserving the following properties:

- Each one of the partitions has the structure described in Section 2.2.2, i.e. it is a reverse map from features to files.

- At ingestion time, the features of each file, f_n, are used to choose m partitions to which the file will be routed. We call m the *routing factor* and $m < K$. The chosen partitions receive *all* the features of the file, *i. e.* $H(f_n)$ is added to each one of the chosen partitions. This algorithm is the *document routing algorithm*.

- At query time, given the query document f_q, the same document routing algorithm is used to choose which partitions to query. The query process for each partition is as described in Section 2.2.2. The chosen partitions are queried in parallel and independently; there is no background communication among them. The results of the queries from the chosen partitions are merged to form the answer to the query.

- Even though we query only a small subset of the partitions (*i. e.* m is much smaller than K) there is minimal loss of recall compared to the case where there is one global index.

We first describe the document routing algorithm and then provide the justification for why it works.

2.3.1 The Document Routing Algorithm

The input to the algorithm is

- $H(f_n)$, the set of features of the document f_n
- an integer K, the number of partitions
- an integer m, the routing factor

We assume that $m < K$, and $|H(f_n)| \geq m$. The feature extraction algorithm, H, extracts features using a min-wise independent hash function as explained in Section 2.2.1. The routing algorithm computes a set of integers $R = \{r_0, r_1, \ldots, r_{m-1}\}$ where $0 \leq i < m$. $r_0, r_1, \ldots, r_{m-1}$ are the partitions to which the document will be routed. The document routing algorithm is as follows:

1. Compute $\text{bot}_m(H(f_n))$ where bot_m is a function that picks the m smallest integers in a set. In other words, for a set of integers S where $|S| \geq m$, $\text{bot}_m(S) \subseteq S$, $|\text{bot}_m(S)| = m$, and $x \in \text{bot}_m(S) \wedge y \in S \Rightarrow x \leq y$.

2. For every hash h in $\text{bot}_m(H(f_n))$ compute $(h \mod K)$. Any other consistent distribution scheme can also be used [43, 56]. R is the set of the resulting integers, $R = \{h \mod K | h \in \text{bot}_m(H(f_n))\}$. The document is now routed to all the partitions indicated by R.

The routing algorithm has been depicted in Figure 2.3.

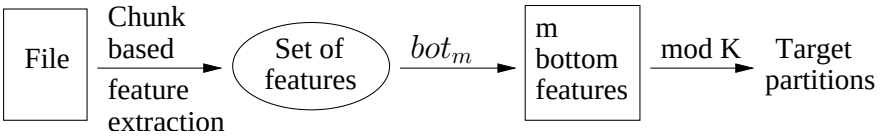

Figure 2.3: Document Routing Algorithm

2.3.2 Why It Works

The routing algorithm is based on a generalization of Broder's theorem [18]. Broder's theorem relies on the notion of a *min-wise independent family of permutations*.

Definition 1 *Let S_n be the set of all permutations of $[n]$. The family of permutations $F \subseteq S_n$ is min-wise independent if for any set $X \subseteq [n]$ and any $x \in X$, when p is chosen uniformly and at random from F we have*

$$Pr(\min\{p(X)\} = p(x)) = \frac{1}{|X|}$$

In practice, truly min-wise independent permutation are expensive to implement. Practical systems use hash functions that approximate min-wise independent permutations.

Theorem 1 *Consider two sets S_1 and S_2, with $H(S_1)$ and $H(S_2)$ being the corresponding sets of the hashes of the elements of S_1 and S_2 respectively, where H is chosen uniformly and at random from a min-wise independent family of permutations. Let $\min(S)$ denote the smallest element of the set of integers S.*

$$Pr(\min(H(S_1)) = \min(H(S_2))) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$$

Broder's theorem states that the probability that the two sets S_1 and S_2 have the same minimum hash element is the same as their Jaccard similarity measure.

Now, consider two files f_i and f_j, and let $m = 1$, i.e. we route each file to only one partition. According to the theorem above, and the definition of similarity measure between two files, the probability that $H(f_1)$ and $H(f_2)$ have the same minimum element is the same as the similarity measure of the two files. In other words, if the two files are very similar, the minimum elements of $H(f_1)$ and $H(f_2)$ are the same with high probability. But if the minimum elements are the same, the two files will be routed to the same partition, since the partition number to which they are routed is the minimum element modulo K, the number of partitions.

While the probability of being routed to the same partition is high when the two files are very similar, the probability drops significantly when the degree of overlap between the two files goes down. For example, if half the features of the two files are the same, and the two files have the same number of features, the Jaccard similarity measure of the two files is $1/3$, i.e. there is only one third chance that they would be routed to the same partition.

To overcome this problem, we route the files to more than one partition, i.e. we choose $m > 1$. The intuitive justification for using the bottom m features for routing is that if by chance a section of a file changes such that the minimum feature is no longer in the set of features, the second least feature will now become the minimum feature with good probability. Our experiments and the theorem below show that with m a modest number (less than 5), we have a very good chance that two files with a fair degree of similarity will be routed to at least one common partition.

To formalize our intuition, we can generalize the Broder theorem as follows:

Lemma 1 *Let S_1 and S_2 be two sets. Let $I = |S_1 \cap S_2|$ and $U = |S_1 \cup S_2|$. Let $B_1 = bot_m(H(S_1))$ and $B_2 = bot_m(H(S_2))$, where H is a min-wise independent hash function.*

Then

$$Pr(B_1 \cap B_2 = \emptyset) \leq \frac{(U-I)(U-I-1)\ldots(U-I-m-1)}{U(U-1)\ldots(U-m-1)}$$

Let $s = I/U$, i.e. s is the Jaccard similarity measure between S_1 and S_2. A good approximation of the above, when m is small and U is large, is

$$Pr(B_1 \cap B_2 = \emptyset) \leq (1-s)^m + \epsilon$$

ϵ is a small error factor in the order of $1/U$.

When we translate this lemma to the case of documents, we get the following:

Corollary 1 *Let f_1 and f_2 be two documents with similarity measure s. When they are each routed to m partitions using the algorithm above, the probability that there will be at least one partition to which both of them are routed is at least $1-(1-s)^m$*

Now, consider the case where the document f_1 has been ingested into the system, and we now wish to use f_2 as the query document to do similarity based retrieval. Let us say that the similarity measure of f_1 and f_2 is $1/3$, and m, the routing factor, is 4. Since the same routing algorithm is used for ingestion and query processes, and for the query to succeed it suffices that at least one partition be in common between the two files, the probability that we find the document f_1 when we query with f_2 is better than 80%. Contrast this with the case where $m = 1$, when the probability of finding f_1 is only 33%.

The following sections discuss the experimental setup and results.

2.4 Experimental Setup

The experimental data set consisted of 179874 files. These were Hewlett Packard's internal support documents in HTML format. There were 1504984 unique features extracted from this set. A randomly selected subset, F_q, of 332 files was chosen from the original corpus to be used as query files. The rest of the files, the set F_d, was our document repository. Our goal was to find for every file $f_q \in F_q$ the files in F_d that were highly similar to f_q. The similarity measure between two files was calculated using their features as explained in Section 2.2.

Since F_d was our document repository every file $f_d \in F_d$ was used to build the partitions using the document routing algorithm as explained in Section 2.2.3. Every file in F_q was then used to query the partitions as explained in Section 2.2.4 to find similar files to itself in F_d. The number of partitions, K, were varied from 1 through 128. The routing factor, m, used to route every file in F_d (for building the partitions) and in F_q (for querying the partitions) was also varied from 1 through 10.

The result set for every query in F_q using a single non-partitioned index was then used as a standard to judge the quality of results produced when the index was partitioned.

2.5 Results

In the first set of experiments we have compared the quality of results obtained when using a single monolithic feature index ($K = 1$) to search for similar files with those obtained when we had multiple partitions ($K > 1$). First, a monolithic feature index was built using every file in F_d. Next, for every query file $f_q \in F_q$ the set of files similar to it were identified by

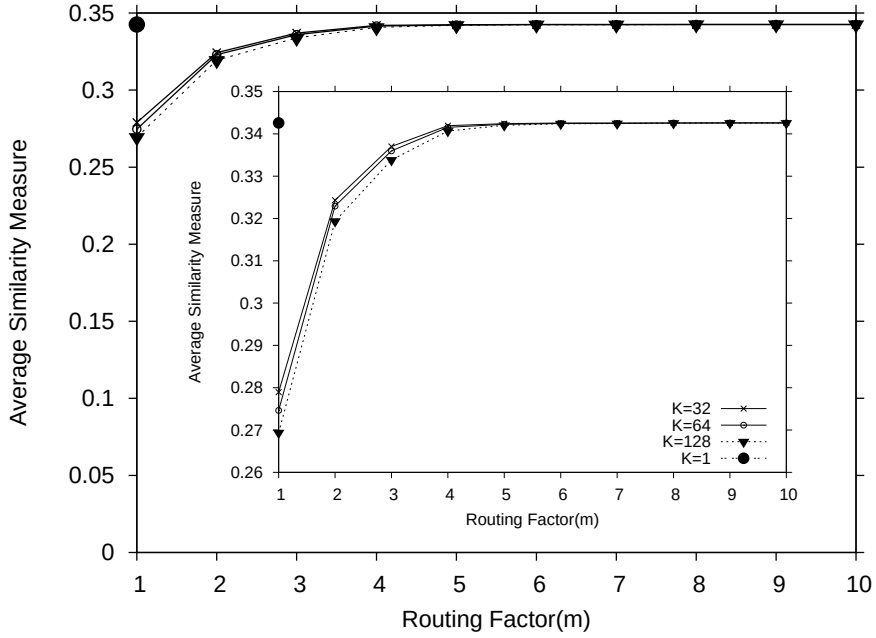

Figure 2.4: Effect of the routing factor on the average similarity measure

querying the monolithic feature index. Each file, f_r, in the result set for every query f_q was then ranked based on its Jaccard similarity index with f_q. The file with the highest similarity measure was the file that was *most* similar to the query file and hence, the best result. The best result, thus calculated, was recorded for every query file. The average similarity measure, calculated as the average of the best results for all f_q, was then calculated for the entire query set F_q.

The next round of experiments was conducted with increasing number of partitions, $K > 1$. For every value of K, the routing factor, m for every file in F_d and F_q was varied from 1 through 10. Once again, the first step was to build the partitions using F_d for the appropriate values of K and m. Using the *same* values for K and m files in F_q were used to query the partitions. The results obtained from the respective partitions were collated and the best result was recorded for every $f_q \in F_q$. The average similarity measure, for every K and m combination, was then calculated for F_q.

Figure 2.4 shows the effect of increasing the number of partitions, K, and the routing factor, m, on the average similarity measure of F_q. The inset shows the same graph with an expanded Y-axis to show the small differences. In the inset, the data point corresponding to $K = 1$ and $m = 1$ corresponds to the average similarity measure for F_q with one monolithic index. This value is 0.342. We can see that for $K = 128$ and $m = 1$ this value is less than 0.27. This is because with increasing number of partitions while using only the minimum feature ($m = 1$) to route the query file it is possible to not find the best match, or the file with the highest similarity measure. In this case, for 213 out of 332 files, or 64% of the query files we did not find the result with the highest similarity measure. When $m = 1$ even a single change that affects

the minimum feature of the query file can prevent us from finding the best match as has been explained in Section 2.3.2.

The overall average similarity measure for F_q, thus, reduces. However, as we increase m, we improve our chances of finding the best result because we now route every file $f_d \in F_d$ and $f_q \in F_q$ to multiple partitions. We can see that even with $m = 3$ there is a significant improvement in the average similarity measure of F_q for all values of K. For $K = 128, m = 3$ this value is more than 0.33. Here, only 51 or 15% of the 332 query files did not find their optimal results. This means that for a large percentage of query files we are being able to find the file in F_d that shares the highest content overlap with them. For $m > 4$ the average similarity measure for all values of K is 0.342 which means that for every query file we were able to find the best match.

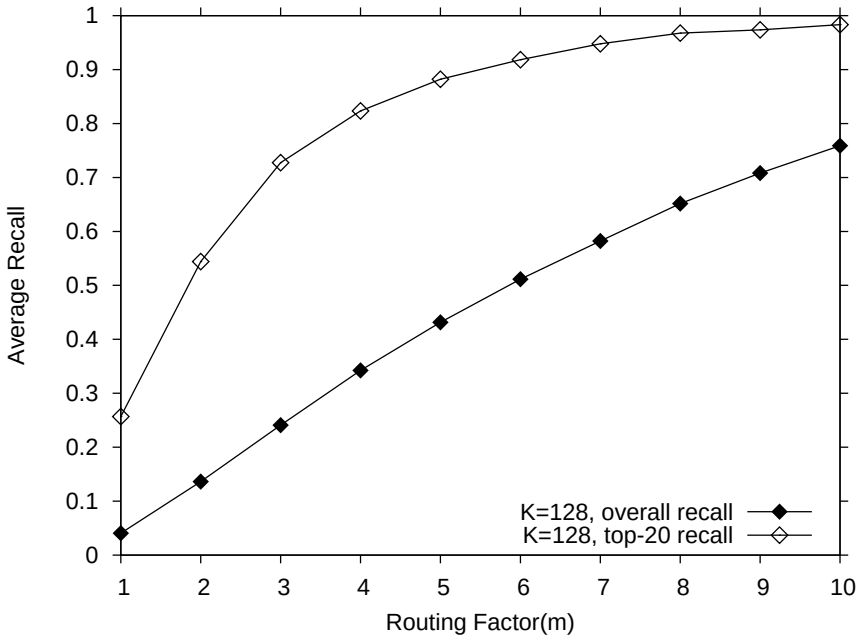

Figure 2.5: Effect of the number of partitions on the overall recall

Figure 2.5 depicts the average recall obtained for all the queries for increasing values of m and for $K = 128$. The recall for every query was calculated as the fraction of the size of the result set obtained when $K > 1$ with respect to the original size of results with $K = 1$. The ideal recall, thus, is 1. The graph in figure 2.5 depicts the average recall for $K = 128$ for increasing values of m in two forms. The first form is the overall recall which takes into account the complete result set obtained for every query. The second form is the recall for only a subset of the result set — specifically the top-20 results in the result set. In this case we retained only the top-20 results for every query. We can see that for $m = 3$ the overall recall is 24% whereas the recall for the corresponding top-20 results is 73%. This result shows us that though with $K = 128$ and $m = 3$ we were able to fetch only 24% of the total result set on an average, this subset contained most of the highest ranking results. This means that we were able to retain and produce the strong resemblances between documents.

We may have lost some of the weak similarity relationships but this loss is acceptable given the gain in scalability. Moreover, for many applications, only the documents with the strongest similarity to the query are of interest, and the low-similarity hits may not be of interest or get filtered out. For example, in the case of a standard search engine users are interested in only the the top few results of their query or the first page of results returned by the search engine. In such a situation the recall achieved by our routing algorithm with respect to the top-20 results is sufficient. However, if an application requires that *every* similar document to a query be found, then one can easily adopt a policy of querying each and every partition instead of just m out of K. Such a scheme will preserve the original recall of every query as was the case when there existed just one index($K = 1$).

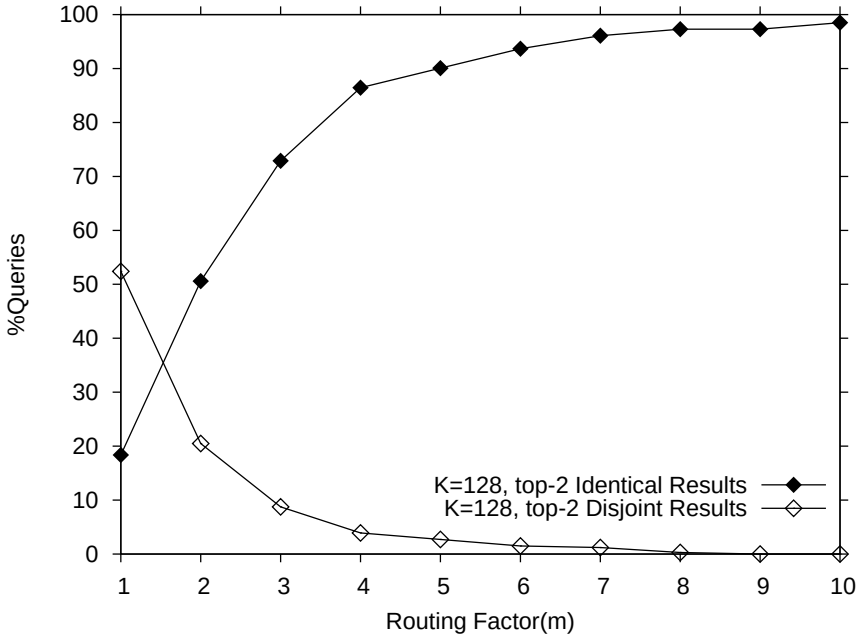

Figure 2.6: Identical and disjoint top-2 lists, 128 partitions

Figure 2.6 shows us exactly how many of the top-2 results with $K = 128$ were identical or disjoint when compared to those with $K = 1$. This data was obtained using techniques developed by Fagin [36]. Identical results were those in which the contents of results were preserved with $K = 128$. Disjoint results were those in which none of the contents of the original top-2 results with $K = 1$ were preserved when $K = 128$. The rest of the top-2 results for $K = 128$ contained at least one of the original top-2 results. We can see that as m increases, even with $m = 3$, more than 70% of the top-2 lists were identical and less than 10% were disjoint. This means that overall more than 90% of the queries returned at least one of their top results.

Figure 2.7 depicts the average partition sizes as compared with the size of the monolithic index for increasing values of m. The average partition sizes have been shown as a

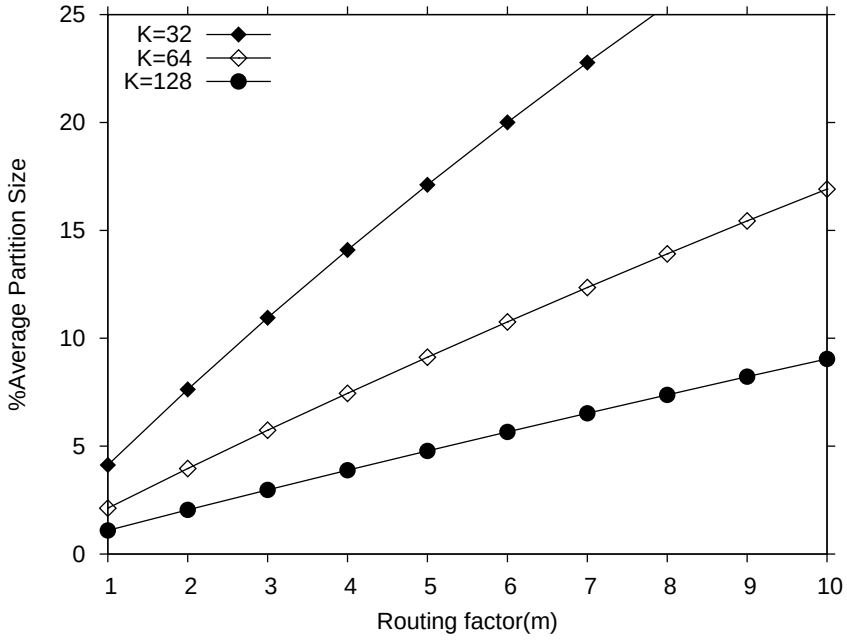

Figure 2.7: %Average partition sizes for increasing values of the routing factor

percentage of the size of the monolithic index. The partition size is the number of keys in the partition — the number of features indexed at every partition. This size does not take into account the list of files that every feature occurs in. Even if we had accounted for it we would observe the same trend as has been shown in the figure. We can see that with $K = 128$ and $m = 3$ the average partition size is less than 3% of the size of the monolithic index.

We have already seen from our previous results that with $K = 128$ and $m = 3$ we obtain very good average similarity for our queries (Figure 2.4) more than 70% of the queries returned their top-2 results identical to when $K = 1$ and more than 90% of the top-2 results contained at least one of the original top-2 results. We can clearly see that our routing algorithm has performed very well while reducing the individual partitions to more manageable sizes,

enabling the parallel execution of similarity based searches while at the same time has not compromised the quality of our results.

2.6 Related Work

Routing keyword queries to promising sources of information has been an active area of research in the field of distributed information retrieval and peer-to-peer networks [22, 24, 26, 51]. Cooper [24] and Lu *et al.* [51] use past information about query results to guide queries to promising sources in peer-to-peer and federated information systems. Distributed Hashing has also been studied widely. Litwin *et al.* [50] proposed Scalable Distributed Data Structures based on linear hash tables for parallel and distributed computing. Distributed Hash Tables(DHT) have also been widely used in the area of peer-to-peer systems to distribute and locate content without having to flood every node in the system with content and queries. Content Addressable Network (CAN) [68], Chord [76], Pastry [71] and Tapestry [86] are some of the DHT implementations used in a distributed environment. Manku [54] has categorized the DHT routing methods into deterministic and randomized. Oceanstore [46] is an infrastructure that provides access to data stored across a large-scale globally distributed system and uses the Tapestry DHT protocol to route queries and place objects close to their access points with the objective of minimizing latency, preserving reliability and maximizing the network bandwidth utilization. PAST [30] is an internet scale global storage utility that uses Pastry's routing scheme. PAST routes a file to be stored to k nodes within the network such that those node identifiers are numerically closest to the file identifier. Pastiche [25] is a peer-to-peer data backup facility that aims to reduce the storage overhead by identifying nodes that share common data at a sub-file granularity. Pastiche

aims to conserve storage space by identifying overlapping content using techniques introduced in the Low-Bandwidth Network File System [57]. In order to route data to appropriate nodes, Pastiche needs to access and maintain an abstract of the file system's contents.

2.7 Summary

The similarity-based search techniques choose, for every file, only a few of its feature hashes to find documents within a large repository highly similar to it. It is based on Broder's theorem which states that the degree of similarity between two sets is proportional to the probability that their minimum hash is the same. Features extracted from every document are stored in an inverse index for fast retrievals. Each feature in the index points to a list of files that it occurs in. To find similar documents, the features of the query document are are sorted and only the top m features are used to query the feature index. m is the routing factor which influences the size of the result. In general, increasing the routing factor identifies more similar documents. However, this also means more number of queries.

The document routing algorithm enables scalable and parallel similarity-based searches. This helps the searches scale gracefully as the repository grows. The feature index is partitioned using distributed placement techniques. How many partitions to query is dictated by the routing factor. Which partitions to query is dictated by the first m features of the document, and, hence, dictated by the contents of the document. Our results show that by querying less than 3% of the partitions the quality of the results is not compromised. It is always possible to increase the routing factor if a more exhaustive search is required.

Chapter 3

Scalable, Parallel Deduplication for Chunk-based File Backup

3.1 Introduction

The amount of digital information created in 2007 was 281 exabytes; by 2011, it is expected to be 10 times larger [38]. 35% of this data originates in enterprises and consists of unstructured content, such as office documents, web pages, digital images, audio and video files, and electronic mail. Enterprises retain such data for corporate governance, regulatory compliance [1, 2], litigation support, and data management.

To mitigate storage costs associated with backing up such huge volumes of data, data deduplication is used. Data deduplication identifies and eliminates duplicate data. Storage space requirements can be reduced by a factor of 10 to 20 or more when backup data is deduplicated [13].

Chunk-based deduplication [37, 57, 66], a popular deduplication technique, first divides input data streams into fixed or variable-length chunks. Typical chunk sizes are 4 to 8 kB. A cryptographic hash or *chunk ID* of each chunk is used to determine if that chunk has been backed up before. Chunks with the same chunk ID are assumed identical. New chunks are stored and references are updated for duplicate chunks. Chunk-based deduplication is very effective for backup workloads, which tend to be files that evolve slowly, mainly through small changes, additions, and deletions [83].

Inline deduplication is deduplication where the data is deduplicated before it is written to disk as opposed to post-process deduplication where backup data is first written to a temporary staging area and then deduplicated offline. One advantage of inline deduplication is that extra disk space is not required to hold and protect data yet to be backed up. Data Domain, Hewlett Packard, and Diligent Technologies are a few of the companies offering inline, chunk-based deduplication products.

Unless some form of locality or similarity is exploited, inline, chunk-based deduplication, when done at a large scale faces what has been termed the *disk bottleneck problem*: to facilitate fast chunk ID lookup, a single index containing the chunk IDs of all the backed up chunks must be maintained. However, as the backed up data grows, the index overflows the amount of RAM available and must be paged to disk. Without locality, the index cannot be cached effectively, and it is not uncommon for nearly every index access to require a random disk access. This disk bottleneck severely limits deduplication throughput.

Traditional disk-to-disk backup workloads consist of data streams, such as large directory trees coalesced into a large file, or data generated by backup agents to conform with legacy

tape library protocols. There are large stretches of repetitive data among streams generated on a daily or weekly basis. For example, files belonging to a user's My Documents directory appear in approximately the same order every day. This means that when files *A*, *B*, *C*, and, thus their chunks, appear in that order in today's backup stream, tomorrow when file *A*'s chunks appear, chunks for files *B* and *C* follow with high probability.

Existing approaches exploit this 'chunk locality' to improve deduplication throughput. Zhu *et al.* [87] store and prefetch groups of chunk IDs that are likely to be accessed together with high probability. Lillibridge *et al.* [49] batch up chunks into large segments, on the order of 10 MB. The chunk IDs in each incoming segment are sampled and the segment is deduplicated by comparing with the chunk IDs of only a few carefully selected backed up segments. These are segments that share many chunk IDs with the incoming segment with high probability.

We now consider the case for backup systems designed to service fine-grained low-locality backup workloads. Such a workload consists of files, instead of large data streams, that arrive in random order from disparate sources. We cannot assume any locality between files that arrive in a given window of time. Several scenarios generate such a workload: File backup and restore requests made by Network-attached Storage (NAS) clients; Continuous Data Protection (CDP), where files are backed up as soon as they have changed; and electronic mails that are backed up as soon as they are received. In absence of locality, existing approaches perform poorly: either their throughput [87] or their deduplication efficiency [49] deteriorates.

Our solution, Extreme Binning, exploits file similarity. Similar files share content and hence are good candidates to deduplicate against each other. However, the key to our solution is to be able to find similar files in a way that is scalable and effective. One cannot do a pairwise

comparison of files using, for example, a unix 'diff' application. This is because such a solution does not scale as the number of files grow. Hence, we borrow from the similarity-based search solution proposed in Chapter 2. We use the concepts of 'set similarity' and Broder's theorem. For every incoming file we need to achieve two objectives: we must find which chunks of the file are duplicates and we need to do this fast. This means that we do not have the bandwidth for executing an exhaustive search to deduplicate every incoming file. To speed up the search, we need to limit our search space. Hence, we do not compare the incoming file's contents with *all* the files in the store. Instead we only compare it with a well chosen subset of files—this subset consists of only those files that we know are similar to the incoming file with high probability. By limiting the search space, we speed up deduplication.

To reduce disk accesses, we split up the chunk index into two tiers. One tier is small enough to reside in RAM and the second tier is kept on disk. Extreme Binning makes a single disk access for chunk lookup per file, thus alleviating the disk bottleneck problem. In the absence of locality, a disk access would have been required per chunk.

We also show how to scale out and parallelize file deduplication. In a distributed setting, with multiple *backup nodes* – nodes that perform file based backup – every incoming file is allocated, using a stateless routing algorithm, to a single backup node only. Backup nodes are autonomous – each node manages its own index and data without sharing or knowing the contents of other backup nodes. To our knowledge, no other approach can be scaled or parallelized as elegantly and easily as Extreme Binning.

One disadvantage of Extreme Binning compared to approaches by Zhu *et al.* [87] and Rhea *et al.* [69] is that it allows some duplicate chunks. In practice, however, as shown by our

experiments, this loss of deduplication is minimal for representative workloads, and is more than compensated for by the low RAM usage and scalability of our approach.

3.2 Chunk-based Deduplication

Chunking divides a data stream into fixed [66] or variable length chunks. Variable-length chunking has been used to conserve bandwidth [57], to weed out near duplicates in large repositories [37], for wide-area distributed file systems [5], and, to store and transfer large directory trees efficiently and with high reliability [35].

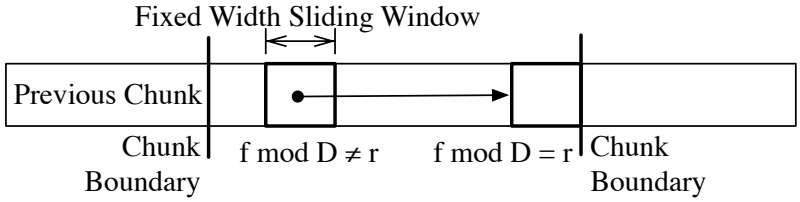

Figure 3.1: Sliding Window Technique

Figure 3.1 depicts the sliding window chunking algorithm. To chunk a file, starting from the beginning, its contents as seen through a fixed-sized (overlapping) sliding window are examined. At every position of the window, a fingerprint or signature of its contents, f, is computed using hashing techniques such as Rabin fingerprints [67]. When the fingerprint, f, meets a certain criteria, such as $f \bmod D = r$ where D, the divisor, and r are predefined values; that position of the window defines the boundary of the chunk. This process is repeated until the complete file has been broken down into chunks. Next, a cryptographic hash or chunk ID of the chunk is computed using techniques such as MD5 [70], or SHA [59, 58].

After a file is chunked, the index containing the chunk IDs of backed up chunks is queried to determine duplicate chunks. New chunks are written to disk and the index is updated with their chunk IDs. A *file recipe* containing all the information required to reconstruct the file is generated. The index also contains some metadata about each chunk, such as its size and retrieval information.

How much deduplication is obtained depends on the inherent content overlaps in the data, the granularity of chunks and the chunking method [65, 31]. In general, smaller chunks yield better deduplication.

3.3 Our Approach: Extreme Binning

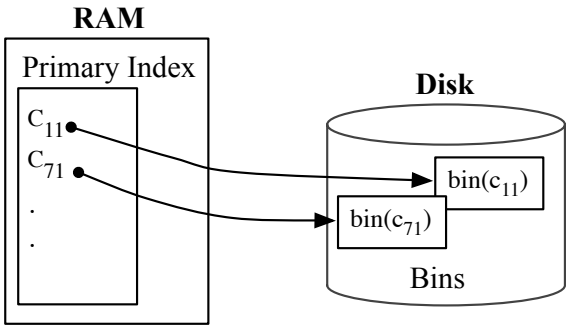

Figure 3.2: A two-tier chunk index with the primary index in RAM and bins on disk

Extreme Binning splits up the chunk index into two tiers. The top tier called the primary chunk index or *primary index* resides in RAM. The primary index contains one chunk ID entry *per file*. This chunk ID is the *representative chunk ID* of the file. The rest of the file's chunk IDs are kept on disk in the second tier which is a mini secondary index that we call *bin*.

Each representative chunk ID in the primary index contains a pointer to its bin. This two-tier index has been depicted in Figure 3.2. The two-tier design distinguishes Extreme Binning from some of the other approaches [87, 69] which use a *flat chunk index* – a monolithic structure containing the chunk IDs of all the backed up chunks.

3.3.1 Choice of the Representative Chunk ID

The success of Extreme Binning depends on the choice of the representative chunk ID for every file. This choice is governed by Broder's theorem [17] as explained in Section 2.3.2.

Broder's theorem states that the probability that the two sets S_1 and S_2 have the same minimum hash element is the same as their Jaccard similarity coefficient [44]. So, if S_1 and S_2 are highly similar then the minimum element of $H(S_1)$ and $H(S_2)$ is the same with high probability. In other words, if two files are highly similar they share many chunks and hence their minimum chunk ID is the same with high probability. Extreme Binning chooses the minimum chunk ID of a file to be its representative chunk ID.

Our previous work [11] has shown that Broder's theorem can be used to identify similar files with high accuracy without using brute force methods. In this previous work, chunk IDs extracted from each file within the given corpus were replicated over multiple partitions of the search index. Each partition was expected to fit in RAM. Such a solution is not feasible in the context of large scale backup systems that can be expected to hold exabytes of data, and are the focus of this work.

Extreme Binning extends our previous work in that it applies those techniques to build a scalable, parallel deduplication technique for such large scale backup systems. Our

contributions in this work are: the technique of splitting the chunk index into two tiers between the RAM and the disk to achieve excellent RAM economy, the partitioning of the second tier into bins, and the method of selecting only one bin per file to amortize the cost of disk accesses without deteriorating deduplication. Together, these contributions extend our previous work considerably.

3.3.2 File Deduplication and Backup using Extreme Binning

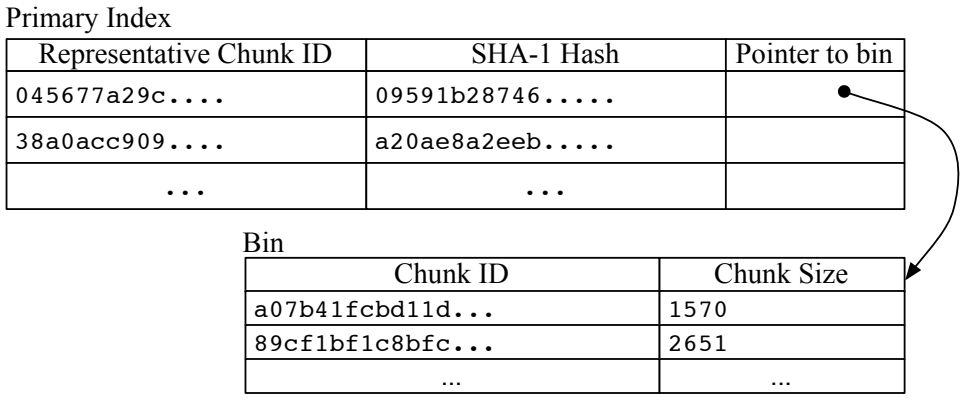

Figure 3.3: Structure of the primary index and the bins

Figure 3.3 shows the structure of the primary index and one bin. There are three fields in the primary index – the representative chunk ID, the whole file hash and a pointer to a bin. Each bin contains two fields: the chunk ID and the chunk size. In addition, bins may also contain other metadata, such as the address of the corresponding chunk on disk. File backup proceeds as follows:

When a file arrives to be backed up, it is chunked, its representative chunk ID is determined and its whole file hash is computed. The whole file hash is a hash of the entire file's contents computed using techniques such as MD5 and SHA-1.

The primary index is queried to find out if the file's representative chunk ID already exists in it. If not, a new secondary index or bin is created. All unique chunk IDs of the file along with their chunk sizes are added to this bin. All the chunks and the new bin are written to disk. The representative chunk ID, the whole file hash and a pointer to this newly created bin, now residing on disk, is added to the primary index.

If the file's representative chunk ID is found in the primary index, its whole file hash is compared with the whole file hash in the primary index for that representative chunk ID. If the whole file hashes do not match, the bin pointer in the primary index is used to load the corresponding bin from disk. Once the bin is in RAM, it is queried for the rest of the file's chunk IDs. If a chunk ID is not found, it is added to the bin and its corresponding chunk is written to disk. Once this process has been completed for all the chunk IDs of the file, the updated bin is written back to disk. The whole file hash in the primary index is *not* updated.

A whole file hash match means that this file is a duplicate file: all its chunks are duplicates. There is no need to load the bin from disk. File deduplication is complete. By keeping the whole file hash in the primary index, we avoid making a disk access for chunk lookup for most duplicate files.

Finally, references are updated for duplicate chunks and a file recipe is generated and written to disk. This completes the process of backing up a file.

The cost of a disk access, made for chunk ID lookup, is amortized over *all* the chunks of a file instead of there being a disk access per chunk. The primary index, since it contains entries for representative chunk IDs only, is considerably smaller in size than a flat chunk index and can reside in RAM. Hence, it exhibits superior query and update performance.

Our experiments show that Extreme Binning is extremely effective in deduplicating files. To explain why Extreme Binning is effective, we need to understand the significance of the binning technique. We know from Broder's theorem that files with the same representative chunk ID are highly similar to each other. By using the representative chunk ID of files to bin the rest of their chunk IDs, Extreme Binning groups together files that are highly similar to each other. Each bin contains chunk IDs of such a group of files. When a new file arrives to be backed up, assuming its representative chunk ID exists in the primary index, the bin selected for it contains chunk IDs of chunks of files that are highly similar to it. Therefore, duplicate chunks are identified with high accuracy.

Only one bin is selected per file, so that if any of the file's chunk IDs do not exist in the selected bin but exist in other bins, they will be deemed as new chunks. Hence, duplicates are allowed. However, Extreme Binning is able to deduplicate data using fewer resources, e.g., less RAM and fewer disk accesses, which translates to high throughput. Extreme Binning, thus, represents a trade off between deduplication throughput and deduplication efficiency. However, our results show that this loss of deduplication is a very small one.

We now discuss how Extreme Binning can be used to parallelize file backup to build a scalable distributed system.

3.4 A Distributed File Backup System using Extreme Binning

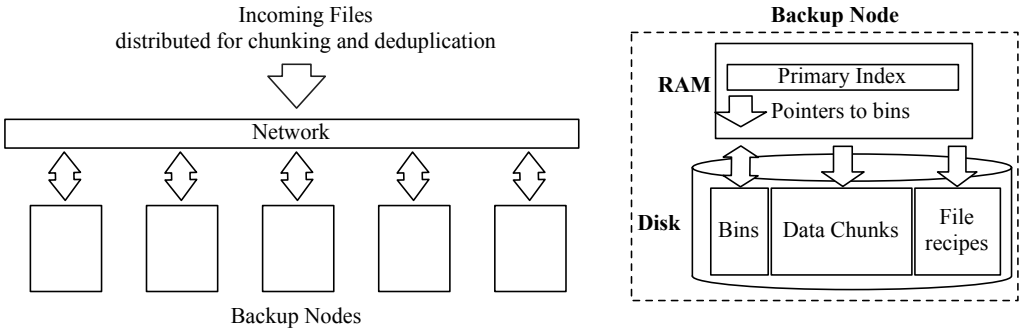

Figure 3.4: Architecture of a Distributed File Backup System build using Extreme Binning

To distribute and parallelize file backup using multiple backup nodes, the two-tier chunk index must first be partitioned and each partition allocated to a backup node. To do this, every entry in the primary index is examined to determine to which backup node it should go. For example, if there are K backup nodes, then every representative chunk ID c_i, in the primary index, is allocated to backup node $c_i \bmod K$. Techniques such as RUSH [43] or LH* [50] can also used for this distribution. These techniques are designed to distribute objects to maximize scalability and reliability.

When a primary index entry moves, the bin attached to it also moves to the same backup node. For every chunk ID in the bin there exists a corresponding data chunk. All the data chunks attached to the bin also move to the same backup node. Each bin is independent. The system has no knowledge of any common chunk IDs in different bins. This means that if a chunk ID appears in two bins, there will be two copies of its corresponding data chunk. Hence, moving a primary index entry to a new backup node, along with its bin and data chunks, does

not create any dependencies between the backup nodes. Even if more backup nodes are added in the future to scale out the backup system, the bins and their corresponding chunks can be redistributed without generating any new dependencies. This makes scale out operations clean and simple.

The architecture of a distributed file backup system built using Extreme Binning has been shown in Figure 3.4. It consists of several backup nodes. Each backup node consists of a compute core and RAM along with a dedicated attached disk. The RAM hosts a partition of the primary index. The corresponding bins and the data chunks are stored on the attached disk as shown in the figure.

When a file arrives to be backed up, it must first be chunked. This task can be delegated to *any one* of the backup nodes, the choice of which, can be based on the system load at that time. Alternatively, a set of master nodes can also be installed to do the chunking. The file is thus chunked by any one of the backup nodes. Its representative chunk ID is extracted and is used to route the chunked file to another backup node – the backup node where it will be deduplicated and stored. The routing is done using the technique described above for partitioning the primary index. If this file is a large file, instead of waiting for the entire file to be chunked, only the first section of the file can be examined to select the representative chunk ID. Note that a file can be chunked by one backup node and deduplicated by another.

When a backup node receives a file to be deduplicated, it uses the file's representative chunk ID to query the primary index residing in its RAM. The corresponding bin, either existing or new, is loaded or created. The primary index and the bin are updated. The updated or new bin, the new chunks, and the file recipe are written to the disks attached to the backup node.

Because every file is deduplicated and stored by only one backup node, Extreme Binning allows for maximum parallelization. Multiple files can be deduplicated at the same time.

Though Extreme Binning allows a small number of duplicates, this number does *not* depend on the number of backup nodes. This loss will be incurred even in a single node backup system. Parallelization does not affect the choice of representative chunk IDs, the binning of chunk IDs, and it does not change what bin is queried for a file. Hence, system scale out does not affect deduplication.

The above distributed design has several advantages. First, the process of routing a file to a backup node is stateless. Knowledge of the contents of any backup node is not required to decide where to route a file. Second, there are no dependencies between backup nodes. Every file – its chunks, index entries, recipe and all – resides entirely on one backup node, instead of being fragmented across multiple nodes. This means that data management tasks such as file restores, deletes, garbage collection, and data protection tasks such as regular integrity checks do not have to chase dependencies spanning multiple nodes. They are clean and simple – all managed autonomously. A file not being fragmented across backup nodes also means better reliability. A fragmented file, whose chunks are stored across multiple backup nodes, is more vulnerable to failures since it depends on the reliability of *all* those nodes for its survival. Autonomy of every backup node, is thus, a highly desirable feature.

3.5 Experimental Setup

Our data sets are composed of files from a series of backups of the desktop PCs of a group of 20 engineers taken over a period of three months. These backups consist all the files for full backups and only modified files for incremental backups. Altogether, there are 17.67 million files containing 162 full and 416 incremental backups in this 4.4 TB data set. We call this data set *HDup* since it contains a high number of duplicates on account of all the full backups. Deduplication of HDup yields a space reduction ratio of 15.6:1.

To simulate an *incremental only* backup workload, we chose from this data set the files belonging to the first full backup of every user along with the files belonging to *all* incremental backups. The first full backup represents what happens when users backup their PCs for the first time. The rest of the workload represents backup requests for files that changed thereafter. This data set is 965 GB in size and consists of 2.2 million files. We call this set *LDup* since it contains few duplicates with a space reduction ratio of 3.33:1.

To evaluate Extreme Binning on a non-proprietary and impartial data set, we have also tested it on widely available Linux distributions. This 26 GB data set consisted of 450 versions from version 1.2.1 to 2.5.75.

It has been shown that data deduplication is more effective when there is low variance in chunk sizes [34]. Consequently, we used the Two Threshold Two Denominators (TTTD) [37] to chunk files. TTTD has been shown to perform better than the basic sliding window chunking algorithm in finding duplicate data and reducing the storage overheads of chunk IDs and their metadata [34]. The average size of the chunks was 4 KB. The chunks were not compressed. Any other chunking algorithm can also be used.

We chose SHA-1 for its collision resistant properties. If SHA-1 is found to be unsuitable, another hashing technique can be used.

Our approach simply enables one to perform scalable efficient searches for duplicate chunks using chunk IDs. Once it has been determined that another chunk with the same chunk ID exists in the backup store, it is always possible to actually fetch the chunk and do a byte by byte comparison instead of a compare by hash [41, 14] (chunk ID) approach, though, at the cost of reduced deduplication throughput as reported by Rhea *et al.* [69].

Our hardware setup was: each backup node ran Red Hat Enterprise Linux AS release 4 (Nahant Update 5). HP MPI (Message-Passing Interface), which contains all the MPI-2 functionality, was used for inter-process communication. Berkeley DB v.4.7.25 [61] was used for the primary index and the bins.

3.6 Results

Extreme Binning was tested for its deduplication efficiency, load distribution, and RAM usage when used to deduplicate the three data sets.

3.6.1 Deduplication Efficiency

Figure 3.5 shows how Extreme Binning did while finding duplicates in HDup. The three curves show how much storage space was consumed when there was no deduplication, if every duplicate chunk was identified (perfect deduplication), and when Extreme Binning was used. A small number of duplicate chunks are allowed by Extreme Binning. With perfect deduplication, the storage space utilization was 299.35 GB (space reduction ratio: 15.16:1),

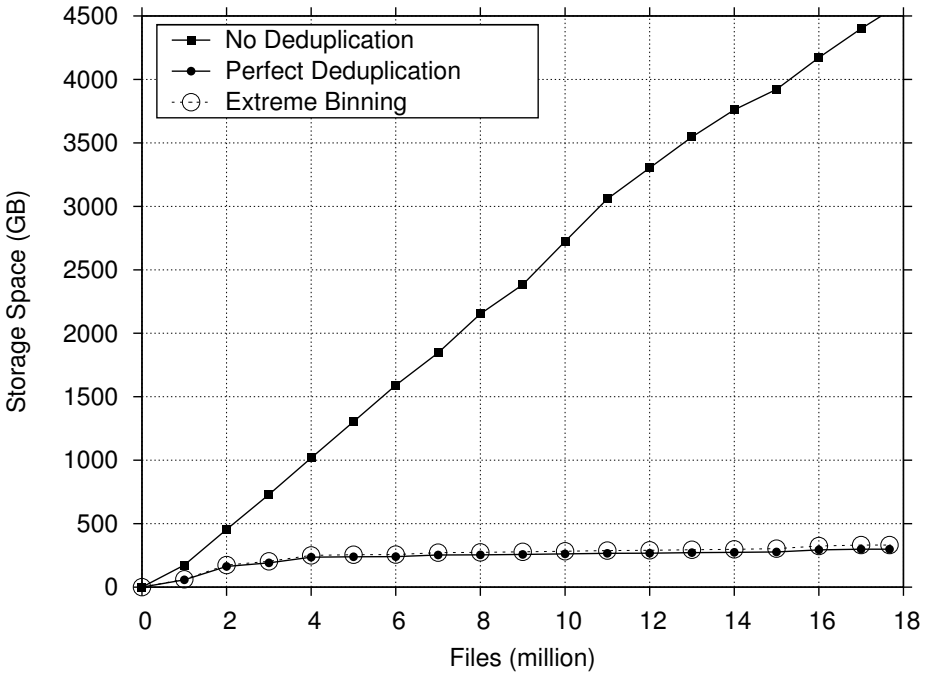

Figure 3.5: Deduplication for HDup

whereas with Extreme Binning it was 331.69 GB (space reduction ratio: 13.68:1). Though Extreme Binning used an extra 32.3 GB, this overhead is very small compared to the original data size of 4.4 TB.

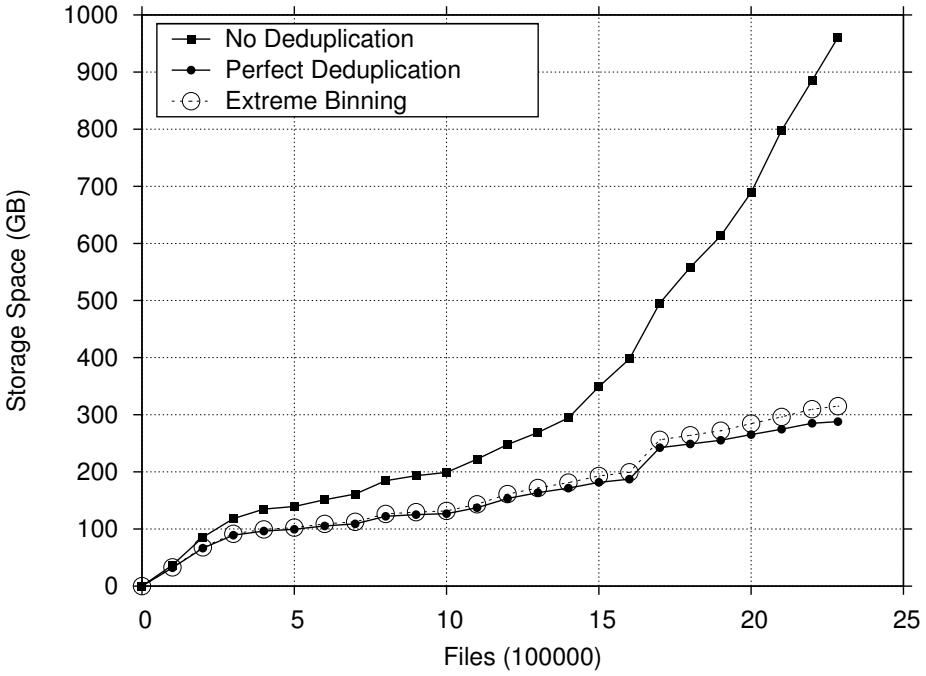

Figure 3.6: Deduplication for LDup

Figure 3.6 shows Extreme Binning's deduplication efficiency for LDup. Perfectly deduplicated, LDup data set required 288.03 GB (space reduction ratio: 3.33:1) whereas Extreme Binning consumed 315.09 GB (space reduction ratio: 3.04:1). From these graphs, it is clear that Extreme Binning yields excellent deduplication and that the overhead of extra storage space is small.

Figure 3.7 shows similar results for the Linux data set. The distributions were ordered according to the version number before being deduplicated. Every point on the x-axis corre-

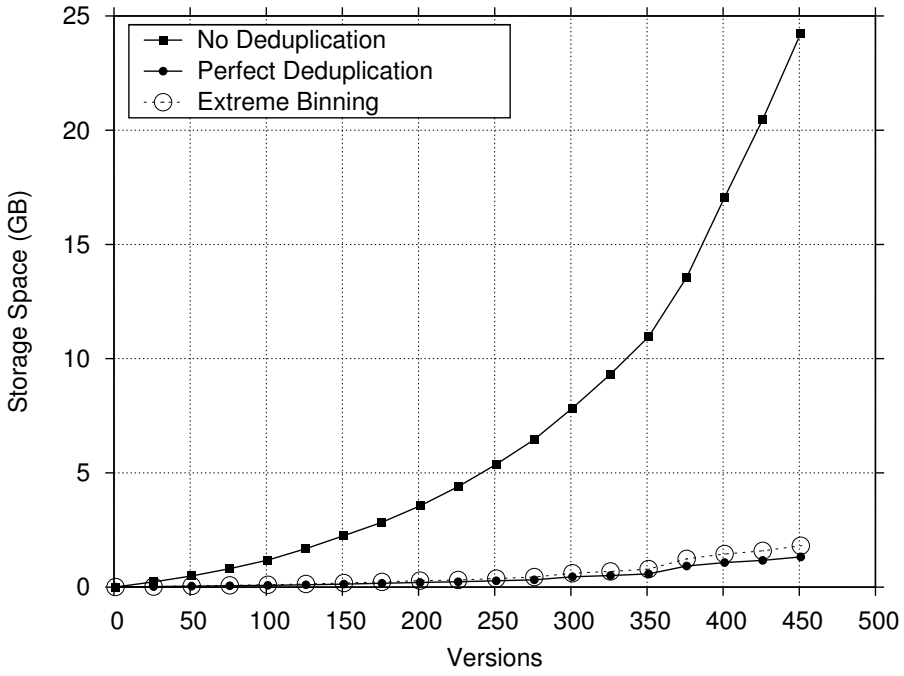

Figure 3.7: Deduplication for Linux

sponds to one version; there are a total of 450 versions. Perfectly deduplicated data size was 1.44 GB (space reduction ratio: 18.14:1) while with Extreme Binning it was 1.99 GB (space reduction ratio: 13.13:1). In this case too, Extreme Binning yields very good deduplication.

3.6.2 Load Distribution

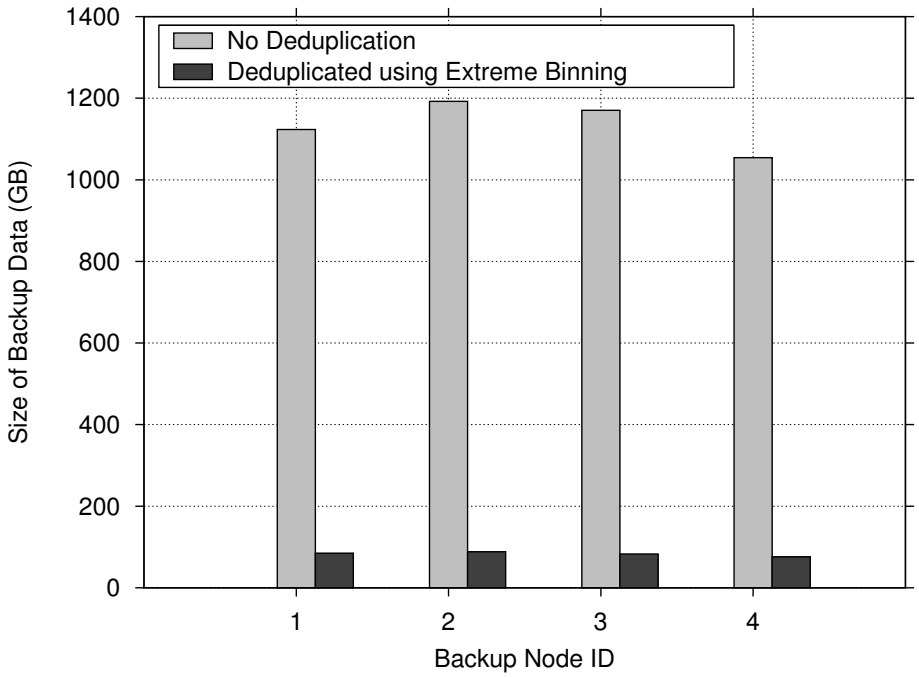

Figure 3.8: Size of backup data on each backup node for HDup when using 4 backup nodes

Figures 3.8 and 3.9 show how much data, both deduplicated and otherwise, is managed by each backup store for HDup and LDup respectively. It is clear that no single node gets overloaded. In fact, the deduplicated data is distributed fairly evenly. The same trend was observed when 2 through 8 nodes were used. This means that the distribution of files to backup

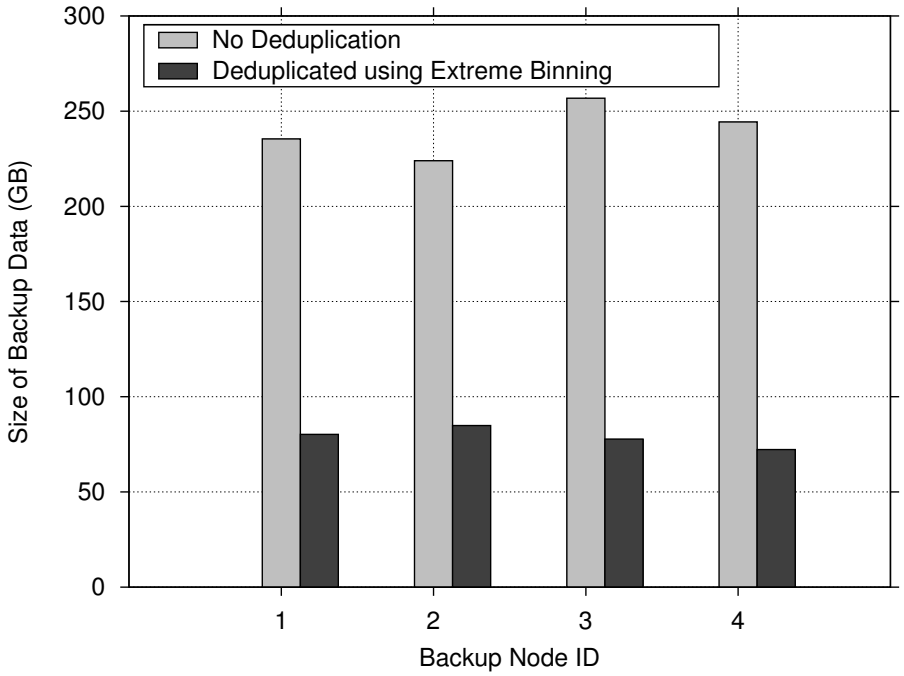

Figure 3.9: Size of backup data on each backup node for LDup when using 4 backup nodes

nodes is not uneven. This property of Extreme Binning is vital towards ensuring smooth scale out and preventing any node from becoming a bottleneck to the overall system performance.

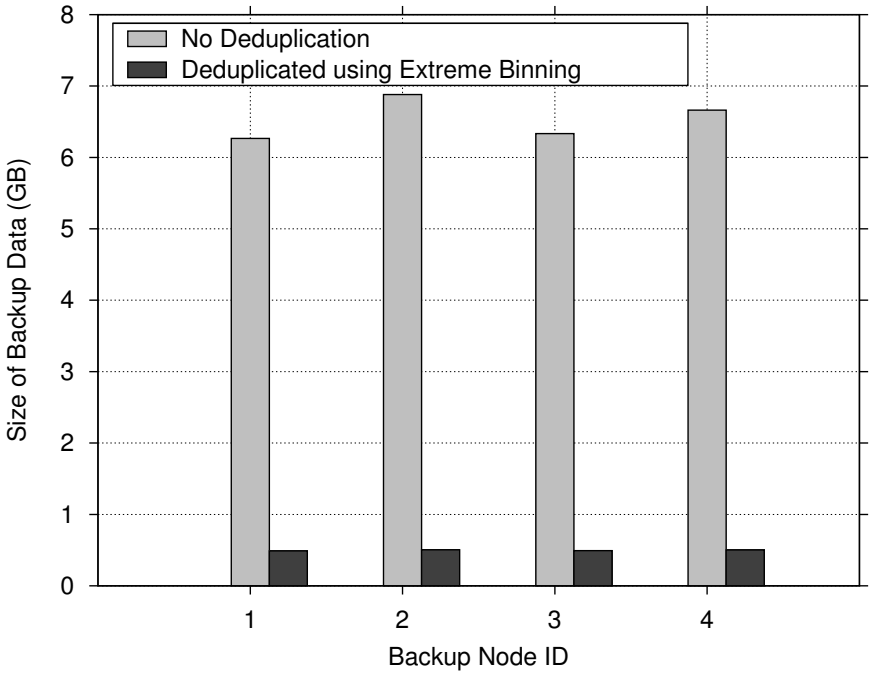

Figure 3.10: Size of backup data on each backup node for Linux when using 4 backup nodes

Figure 3.10 depicts the load distribution of Extreme Binning for Linux when 4 nodes were used. Once again, the same trend was observed when 2 through 8 nodes were used.

3.6.3 Comparison with Distributed Hash Tables

A flat chunk index could be partitioned like a DHT; by using a consistent hashing scheme to map every chunk ID to a partition. Every partition can then be hosted by a dedicated compute node. To deduplicate a file, the same hashing scheme can be used to dictate which partition should be queried to ascertain if the corresponding chunk is duplicate. Assume a file

containing n chunks is to be deduplicated and that the flat chunk index has been partitioned into P partitions such that $n > P$. In the worst case, *all* P partitions will have to queried to deduplicate this file. Our experiments using the DHT technique have shown that for HDup, with $P = 4$, for 52% of the files more than 1 partition was queried and for 27% all 4 partitions were queried. For Linux distributions too, for 50% of the files more than 1 partition was queried and for 12% all 4 partitions were queried. Such a wide query fanout to deduplicate a file reduces the degree of parallelization – fewer files can be deduplicated at the same time. Further, such an infrastructure cannot be used to design a decentralized backup system where every backup node autonomously manages the indices and the data. It is not clear *which* node in the DHT-like scheme stores the deduplicated file given that the file's duplicate chunks may spread across more than one node. Autonomy of backup nodes is not possible in such a design. Further, partitioning the flat chunk index does not reduce RAM requirements. The total RAM usage remains the same whether the index is partitioned or not. Because Extreme Binning only keeps a subset of all the chunk IDs in RAM, its RAM requirement is much lower than a flat chunk index. This means that fewer backup nodes will be required by Extreme Binning than a DHT like scheme to maintain throughput.

3.6.4 RAM Usage

Each primary index entry consisted of the representative chunk ID (20 bytes), the whole file hash (20 bytes), and a pointer to its corresponding bin. The size of the pointer will depend on the implementation. For simplicity we add 20 bytes for this pointer. Then, every record in the primary index is 60 bytes long. With only one backup node, the RAM usage

for Extreme Binning was 54.77 MB for HDup (4.4 TB). Even with the overheads of the data structure, such as a hash table used to implement the index, the RAM footprint is small. For the same data set a flat chunk index would require 4.63 GB. The ratio of RAM required by the flat chunk index to that required by Extreme Binning is 86.56:1. For LDup this ratio is 83.70:1, which proves that Extreme Binning provides excellent RAM economy.

Though the flat chunk index can easily fit in RAM for HDup and LDup, we must consider what happens when the backed up data is much larger. Consider the case of a petabyte of data. If the average file size is 100 kB and the average chunk size is 4 kB, there will be 10 billion files and 250 billion chunks. If the deduplication factor is 15.6, as in the case of HDup, 16 billion of those chunks will be unique, each having an entry in the flat chunk index. Given that each entry takes up 60 bytes, the total RAM required to hold all of the flat chunk index will be 895 GB, while for Extreme Binning this will be only 35.82 GB. For LDup, because of fewer duplicates, the flat chunk index would need over 4 TB of RAM, while Extreme Binning would require 167 GB. Even if the flat index is partitioned, like a DHT, the total RAM requirement will not change, but, the number of nodes required to hold such a large index would be very high. Extreme Binning would need fewer nodes. These numbers prove that by splitting the chunk index into two tiers, Extreme Binning achieves excellent RAM economy while maintaining throughput – one disk access for chunk lookup per file.

3.7 Related Work

Two primary approaches have been previously proposed for handling deduplication at scale: sparse indexing [49] and that of Bloom filters with caching [87].

Sparse indexing, designed for data streams, chunks the stream into multiple megabyte segments, which are then lightly sampled (e.g., once per 256 KB) and the samples are used to find a few segments that share many chunks. Obtaining quality deduplication here is crucially dependent on chunk locality, where each chunk tends to appear together again with the same chunks. Because our use cases have little or no file locality sparse indexing would produce unacceptably poor levels of deduplication for them.

Zhu *et al.*'s approach [87] always produces perfect deduplication but relies heavily on inherent data locality for its cache to be effective to improve throughput. This approach uses an in-memory Bloom filter [15] and caches index fragments, where each fragment indexes a set of chunks found together in the input. The lack of chunk locality renders the caching ineffectual and each incoming new version of an existing chunk requires reading an index fragment from disk.

Foundation [69] is a personal digital archival system that archives nightly snapshots of user's entire hard disk. By archiving snapshots or disk images, Foundation preserves all the dependencies within user data. Our approach is distinct from Foundation in that Extreme Binning is designed to service fine grained requests for individual files rather than nightly snapshots.

However, what sets Extreme Binning decisively apart from all these approaches is that it is parallelizable. It is not clear how to parallelize any of these systems in order to obtain better throughput and scalability.

DEDE [23] is a decentralized deduplication technique designed for SAN clustered file systems that support a virtualization environment via a shared storage substrate. Each host maintains a write-log that contains the hashes of the blocks it has written. Periodically, each host

queries and updates a shared index for the hashes in its own write-log to identify and reclaim duplicate blocks. Deduplication is done out-of-band so as to minimize its impact on file system performance. Extreme Binning, on the other hand, is designed for in-line deduplication and, in a distributed environment the backup nodes do not need to share any index between them. Rather, each backup node deduplicates files independently, using its own primary index and bins only, while still being able to achieve excellent deduplication.

Chunk-based storage systems detect duplicate data at granularities that range from entire file, as in EMC's Centera [33], down to individual fixed-size disk blocks, as in Venti [66] and variable-size data chunks as in LBFS [57]. Variable-width chunking has also been used in the commercial sector, for example, by Data Domain and Riverbed Technology. Deep Store [84] is a large-scale archival storage system that uses three techniques to reduce storage demands: content-addressable storage [33], delta compression [4, 29] and chunking [57].

Delta compression with byte-by-byte comparison, instead of hash based comparison using chunking, has been used for the design of a similarity based deduplication system [6]. Here, the incoming data stream is divided into large, 16 MB blocks, and sampled. The samples are used to identify other, possibly similar blocks, and a byte-by-byte comparison is conducted to remove duplicates.

Distributed Hashing is a technique for implementing efficient and scalable indices. Litwin *et al.* [50] proposed Scalable Distributed Data Structures based on linear hash tables for parallel and distributed computing. Distributed Hash Tables (DHT) have also been widely used in the area of peer-to-peer systems [25] and large-scale distributed storage systems [46, 30] to distribute and locate content without having to perform exhaustive searches across every node

in the system. Chord [76], Pastry [71] and Tapestry [86] are some of the DHT implementations used in a distributed environment.

3.8 Summary

We have introduced a new method, Extreme Binning, for scalable and parallel deduplication, which is especially suited for workloads consisting of individual files with low locality. Existing approaches which require locality to ensure reasonable throughput perform poorly with such a workload. Extreme Binning exploits file similarity instead of locality to make only one disk access for chunk lookup per file instead of per chunk, thus alleviating the disk bottleneck problem. It splits the chunk index into two tiers resulting in a low RAM footprint that allows the system to maintain throughput for a larger data set than a flat index scheme. Partitioning the two tier chunk index and the data chunks is easy and clean. In a distributed setting, with multiple backup nodes, there is no sharing of data or index between nodes. Files are allocated to a single node for deduplication and storage using a stateless routing algorithm – meaning it is not necessary to know the contents of the backup nodes while making this decision. Maximum parallelization can be achieved due to the one file-one backup node distribution. Backup nodes can be added to boost throughput and the redistribution of indices and chunks is a clean operation because there are no dependencies between the bins or between chunks attached to different bins. The autonomy of backup nodes makes data management tasks such as garbage collection, integrity checks, and data restore requests efficient. The loss of deduplication is small and is easily compensated by the gains in RAM usage and scalability.

Chapter 4

Unified Deduplication

4.1 Introduction

Unified Deduplication is a term we use to describe a deduplication engine that can deduplicate *across* a variety of workloads. Figure 4.1 shows such a setup. It shows a deduplication engine servicing a variety of clients. VTL clients send tape images, mail servers send electronic mail messages, NAS boxes and backup agents installed on PCs send a mixture of files/folders. Such a unified deduplication engine is useful because it simplifies management. Organizations do not need to install and administer a separate solution for each of their backup/archival needs—whether it is VTL, NAS or electronic mail. Such a solution is also useful because it can deduplicate *across* different workloads, yielding better storage space savings. For example, consider an electronic mail attachment saved on the desktop of an employee. This attachment will occur both as a part of the exchange server's workload as well the weekly/nightly backups generated by a VTL client. A NAS box hosting an archival folder may have this file as well. A unified deduplication solution can detect such duplicates across

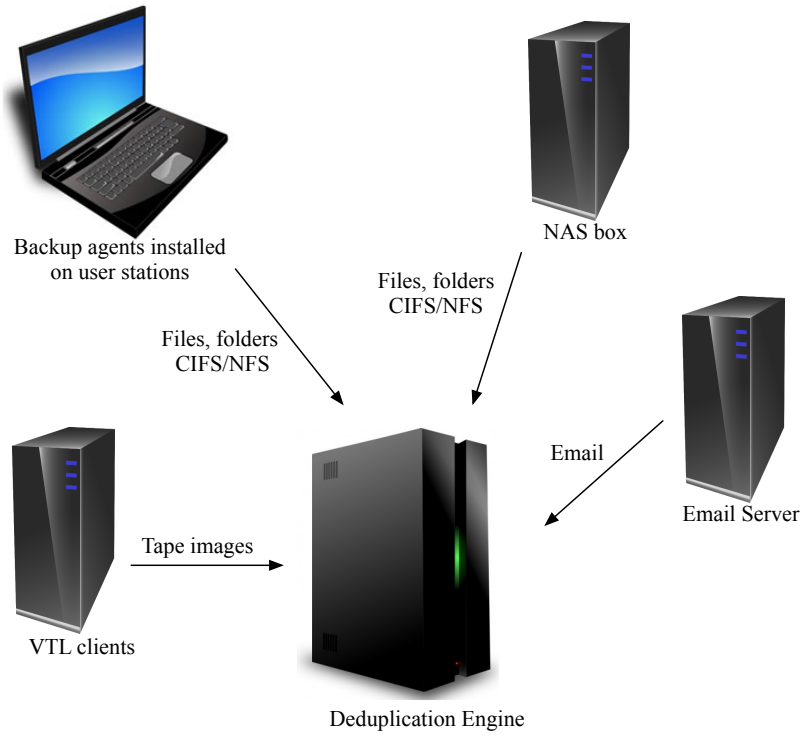

Figure 4.1: Deduplication ratio when using prefix-hash and min-hash sampling

workloads. This is not possible when a dedicated backup/archival deduplication solution has been deployed for each workload. These advantages, of ease of manageability/administration and storage space savings, make unified deduplication a desirable solution.

Implementing a unified deduplication solution is a challenging task. Each workload is unique in its characteristics. Tape images are large files or byte streams on the order of several hundred gigabytes. Individual files and electronic messages vary in size, but are expected to be much smaller than a tape image. They also differ on the amount of locality within their content. There are separate deduplication solutions for each, but so far no unified solution that works well for all situations.

4.2 Workloads, Traditional and Non-traditional

The traditional backup workload consists of large byte streams containing high locality. Tape images generated by virtual tape libraries, large directory nightly snapshots generated using utilities such as 'tar' are examples of such a workload. State-of-the-art technologies such as Sparse Indexing [49] and Locality Sensitive Caching [87] exploit the high locality in such traditional workloads to alleviate the disk bottleneck for maintaining throughput.

Non-traditional backup workloads are made up of files of varying sizes with unreliable locality. Files arrive from disparate sources and there is no locality among objects arriving within a given window of time. Disk-to-disk backup system are fast gaining popularity and deduplication has been incorporated into file-systems [85, 42, 20]. Backup agents installed on NAS boxes periodically send new and modified files and directories to be backed up. Electronic mail messages are sent to be backed up as soon as they arrive. These objects are of much

smaller granularity than streams and lack locality. Extreme Binning [10] exploits file similarity to deduplicate such non-traditional file-based backup workloads yielding excellent deduplication ratios.

Each solution exploits the salient features of its own workload to maximize deduplication quality and throughput. However, there is no unified solution that works well for all situations. Sparse Indexing and Extreme Binning, both, use a sampling approach to reduce RAM usage. Both aim to amortize disk accesses by deduplicating groups of chunks at a time. However, neither can deduplicate the others' workload while still maintaining their deduplication quality. Extreme Binning, which extracts only one sample per object, cannot preserve the deduplication quality when presented with large byte streams. Similarly, Sparse Indexing does poorly when deduplicating small objects.

While it is not possible for Extreme Binning to deduplicate traditional workloads because it extracts only one sample per object, we have found Sparse Indexing to be more adaptable. Hence, we extend Sparse Indexing to build a 'unified deduplication' solution. The objective is to evaluate whether Sparse Indexing can be adapted to use as a unified deduplication solution. How well does it deduplicate files? Can it deduplicate across workloads? We evaluate its performance on the basis of its deduplication quality and RAM usage. We use a combination of traditional and non-traditional workloads. We also experiment with two sampling methods: min-hash and prefix-hash.

4.3 Sampling Chunk Hashes

To sample the chunk hashes of any object is to pick, from the entire set of hashes, certain hashes that obey predefined constraints. The prefix-hash method, used in Sparse Indexing at this time, picks only those hashes whose first n bits are zero. n defines the sampling rate of $1/2^n$. Thus, on average, one out of 2^n hashes are chosen per object. A 10 MB object, for example, yields 2560 4K chunks. A sampling rate of $1/2^7$ yields, on average, 20 samples for this 10 MB object using the prefix-hash method. Since this is probabilistic, there may not always be 20 samples per object—there may not be 20 hashes whose first n bits are zero.

Min-hash, or max-hash, sampling method is one in which all the chunk hashes of the given object are first sorted and then the top m hashes, as defined by the sampling rate, are chosen. Here, m=(number of hashes $\times$ sampling rate). For the 10 MB object and a sampling rate of $1/2^7$, min-hash sampling yields *exactly* 20 samples.

Apart from the actual sampling process, this is the key difference between prefix-hash and min-hash sampling. Min-hash sampling always yields as many samples as defined by the sampling rate, whereas prefix-hash may or may not. This difference is not very significant when objects are large in size, on the order of megabytes. However, for small objects, on the order of 100 KB, there is a high probability that they will not contain the expected number of samples when prefix-hash is used. There is another advantage to min-hash sampling. If one wants to enforce a particular number of samples per object, irrespective of size, it can be done using min-hash and not with prefix-hash. Min-hash, therefore, is more adaptable.

4.4 Sparse Indexing

A concise description of this approach is given here. For a thorough study we refer to the work of Lillibridge *et al.* [49]. Under the sparse indexing approach, *segments* are the unit of storage and retrieval. A block diagram of the process can be found in Figure 4.2. A segment is a sequence of chunks. Data streams are broken into segments in a two step process: first, the data stream is broken into a sequence of variable-length chunks using a chunking algorithm, and, second, the resulting chunk sequence is broken into a sequence of segments using a segmenting algorithm. Segments are usually on the order of a few megabytes. Two segments are similar if they share a number of chunks.

The Two-Threshold Two-Divisor (TTTD) chunking algorithm [34] is used to subdivide the incoming data stream into chunks. TTTD produces variable-sized chunks with smaller size variation than other chunking algorithms, leading to superior deduplication. To avoid the segment boundary-shifting problem, the segmentation algorithm, *variable-size segmentation*, uses the same trick used at the chunking level: the boundaries are based on landmarks in the content, not distance. Variable-size segmentation operates at the level of chunks rather than bytes and places segment boundaries only at existing chunk boundaries. Every Segment is represented in the store using its *manifest*: a manifest or segment recipe [80] is a data structure that allows reconstructing a segment from its chunks. A segment's manifest records its sequence of chunks, giving for each chunk its hash, where it is stored on disk, and possibly its length. Every stored segment has a manifest that is stored on disk.

Incoming segments are deduplicated against similar, already existing segments in the store. To identify similar segments, the chunk hashes of the incoming segment are sampled,

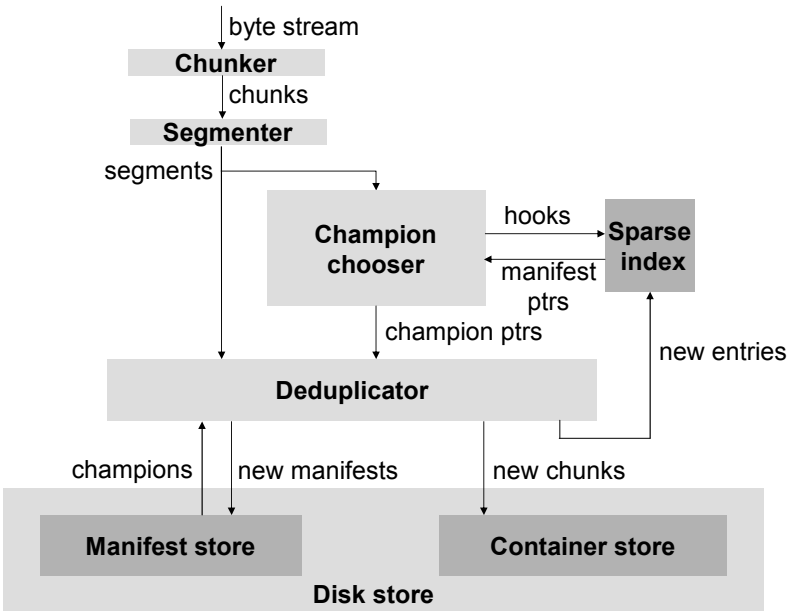

Figure 4.2: Block diagram of the deduplication process

and an in-RAM index is used to determine which already stored segments contain how many of those hashes. The in-memory index, called the *sparse index*, maps hooks to the manifests in which they occur—it keeps pointers to the manifests on disk.

Looking up the hooks of an incoming segment S in the sparse index results in a possible set of manifests against which that segment can be deduplicated. Since loading manifests from disk is costly, only a few well chosen manifests—the *champions*—are fetched from disk. Champions one at a time, until the maximum allowable number of champions are found, or there are no more candidate manifests. Each time the manifest with the highest non-zero score, where a manifest gets one point for each hook it has in common with S that is not already present in any previously chosen champion, is chosen. If there is a tie, the manifest most recently stored is chosen. The choice of which manifests to choose as champions is done based solely on the hooks in the sparse index; that is, it does not involve any disk accesses.

The hashes of the chunks in the incoming segment are then compared with the hashes in the champion manifests to find duplicate chunks. Use of the SHA1 hash algorithm [59] makes false positives extremely unlikely. Those chunks that are found not to be present in any of the champions are stored on disk in chunk containers, and a new manifest is created for the incoming segment. The new manifest, written to disk, contains the location of each incoming chunk on disk.

Finally, the sparse index is updated with an entry for this manifest. New hooks are added and existing hooks are updated with a pointer to this manifest. To conserve space, a maximum limit is set for the number of manifests that can be pointed to by any one hook. If

the maximum is reached, the oldest manifest is removed from the list before the newest one is added.

There is no full chunk index in this approach, either in RAM or on disk. The only index in RAM is the sparse index, which is much smaller than a full chunk index: for example, if only one out of every 128 hashes is sampled, then the sparse index can be 128 times smaller than a full chunk index. The cost of a handful random disk accesses per segment in order to load in champion manifests, is amortized over the thousands of chunks in each segment, leading to acceptable throughput. This is how the chunk-lookup disk bottleneck is avoided..

4.5 Experimental Setup

The first data set is called the *Workgroup* data set. It is composed of files from a series of backups of the desktop PCs of a group of 20 engineers taken over a period of three months. These backups consist of all the files for full backups and only modified files for incremental backups. Altogether, there are 17.67 million files containing 157 full and 409 incremental backups. The total size was 4.4 TB with the average file size being 25 kB.

The second data set is called the *SE3D* data set. It consists of semi-regular copies of the work directories of 11 groups of animators taken during an animation showcase called SE3D. There are a total of 448 versions of directories copied over a period of 10 months. The animators used a commercial content creation application called ®Maya [7] to create digital models that defined their 3D animated movie, including the shape and movement of characters, backgrounds and objects, and associated textures, lighting, and camera definitions. Maya uses over a dozen file formats including a variety of image formats (e.g., JPG and TIFF) and several

proprietary formats; most of these are binary formats, although a few are ASCII (e.g., the MEL scripting language). This data set was 203 GB in size with the average file size being 1.2 MB.

The objective of the experiments is to determine if Sparse Indexing can be used for unified deduplication. To do this, we use Sparse Indexing to deduplicate high locality stream-based workloads, low locality file-based workloads and combination workloads made up of a mixture of these two. We will also determine whether one sampling method is better than the other in terms of its adaptability and the deduplication ratio yielded when using it.

For each data set we generated three workloads:

4.5.1 Files

This workload consisted of a series of individual files within the user directories. Files were not coalesced. For the Workgroup data set, for each user's backup, its files were ordered alphabetically based on their relative path and name. These 'backup groups' were then ordered by the time the backup was taken. For the SE3D data set, for each directory version, its files were also ordered alphabetically. These ordered lists of files were further ordered by the time the directory versions were recorded. Such an ordered list of files constituted the Files workload for Workgroup and SE3D. Each file was chunked separately. For every file, the first chunk began at the beginning of the file, and the last chunk ended with the file. No chunk boundaries cross over file boundaries. This workload represents a file-based workload generated by a NAS box or electronic mails sent by mail servers. Files are deduplicated one file at a time and not in bulk. Hence, to Sparse Indexing, every file was a segment. No locality is assumed between files that arrive within a given window of time. Figure 4.3 shows such a workload.

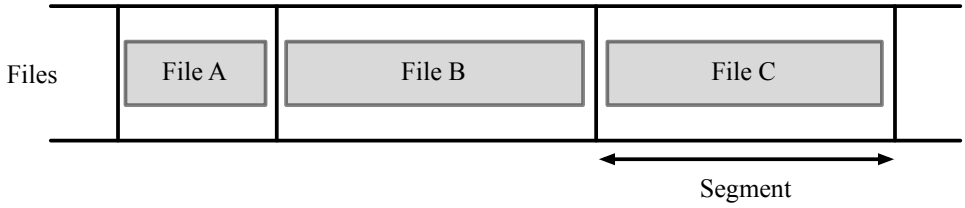

Figure 4.3: Files workload

4.5.2 Streamed Archives

This workload consists of a series of archived files—individual files and directory trees concatenated using the 'tar' software. The archived files were *not* compressed. For the Streamed Archives data set, one archive file was generated for each of the 20 engineer's full and incremental backups. The files within every archive were ordered alphabetically by their path and name. An ordered list of these 566 archive files, ordered by the time that each backup was taken, made up the Streamed Archives workload for Workgroup. For the SE3D data set, one archive file was generated for each of the 448 directory versions. The files and directories within the archive were ordered alphabetically. The list of archives was then ordered by the time that the directory contents were recorded. For both the data sets, each archive file was chunked separately: the first chunk began at the beginning of the archive and the last chunk ended with the archive. No chunk boundary crossed over an archive boundary to include a piece of another archive. Within an archive, a chunk could contain parts of two files coalesced together or even parts of metadata along with content. This is because the chunking program does not use any format specific parser to interpret the contents of the archive. This workload

represents a traditional stream-based workload generated by a virtual tape library where weekly full and daily incremental backups are taken by concatenating files using a 'tar' like software and then streamed to the deduplication engine. Figure 4.4 shows a Streamed Archive workload.

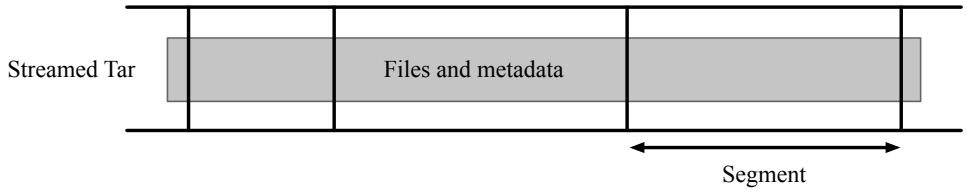

Figure 4.4: Streamed Archives workload

4.5.3 Streamed Chunked Files

This workload consisted of the same series of files as in the Files workload except that these files were concatenated together as follows: each file was first chunked, so the chunk boundaries respected file boundaries. One additional chunk, containing this file's metadata, was created. The metadata consisted of the file path and name, size, access date and time, owner and permissions. A chunked file, then, consisted of the chunks of its contents followed by its metadata chunk. A stream of such chunked files formed the Streamed Chunked Files workload as shown in Figure 4.5. This workload also represents a traditional stream-based workload, but one that has been generated by a client that is aware of and obeys file boundaries when chunking them. Such a client, installed across NAS boxes crawls through user directories or application folders, that are either specially marked or selected based on some criteria, chunks files and generates their metadata chunks. Periodically, it groups together such chunked files and then

transmits a stream of chunks to the deduplication engine. We have chosen to experiment with this workload to study the effect of two factors on deduplication: locality and file boundary recognition. By streaming files one after the other, locality is preserved. By chunking one file at a time and by demarcating the metadata chunks, file boundaries are respected. In a Streamed Archive workload, locality is preserved because files are coalesced, but as the chunking software is not privy to the positions of metadata and file boundaries, chunk boundaries may cross over file and metadata boundaries.

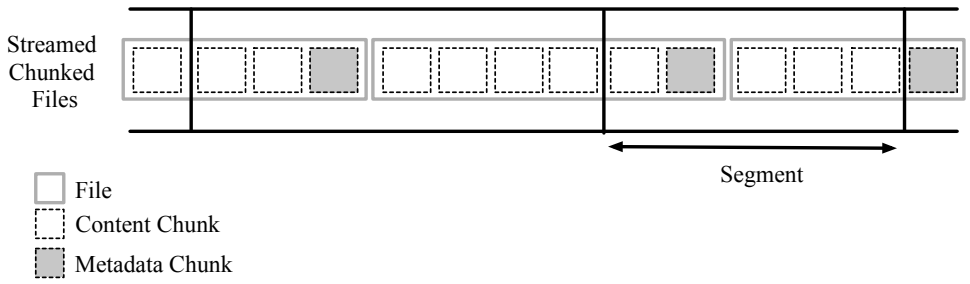

Figure 4.5: Streamed Chunked Files workload

In addition to these, a few 'combination' workloads were also generated.

The combination workloads test whether Sparse Indexing can find duplicates across workloads that contain duplicate data packaged in different forms. They contain a mixture of streamed (both Streamed Archives and Streamed Chunked Files) and Files workloads. Such workloads represent systems where files are packaged into streams, alike traditional backup workloads, and some of the same files may also be transmitted on a file-by-file basis. For example, consider a setup where files are packaged every weekend into a streamed archive for

a full backup, and during the weekdays, only files that change are transmitted individually for incremental backups.

4.6 Results

Various workloads were deduplicating using Sparse Indexing to study its efficacy. The two sampling methods—min-hash and prefix-hash—were also compared.

4.6.1 Deduplicating Streamed Archives, Streamed Chunked Files, Files

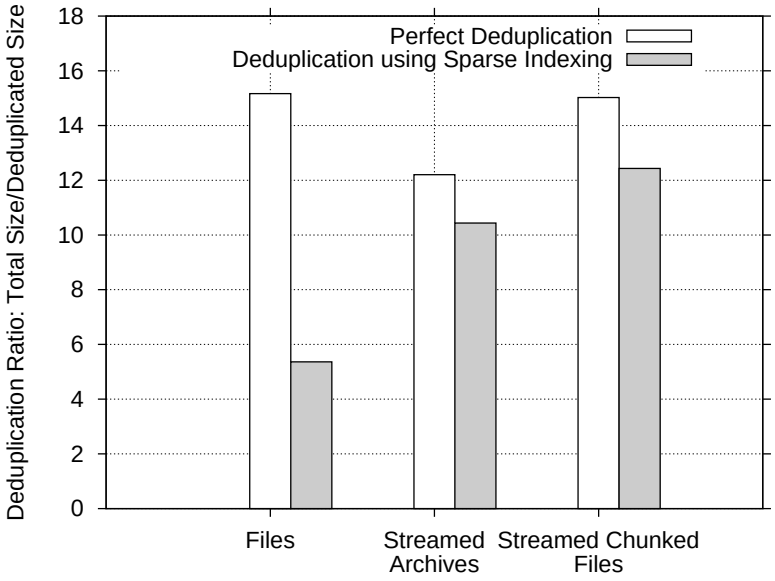

Figure 4.6: Deduplication for Workgroup using min-hash sampling for Files, Streamed Archives, Streamed Files

Figures 4.6 and 4.7 depict the deduplication ratio for the Workgroup and SE3D data sets when using Sparse Indexing. This is compared with the perfect deduplication ratio. Perfect

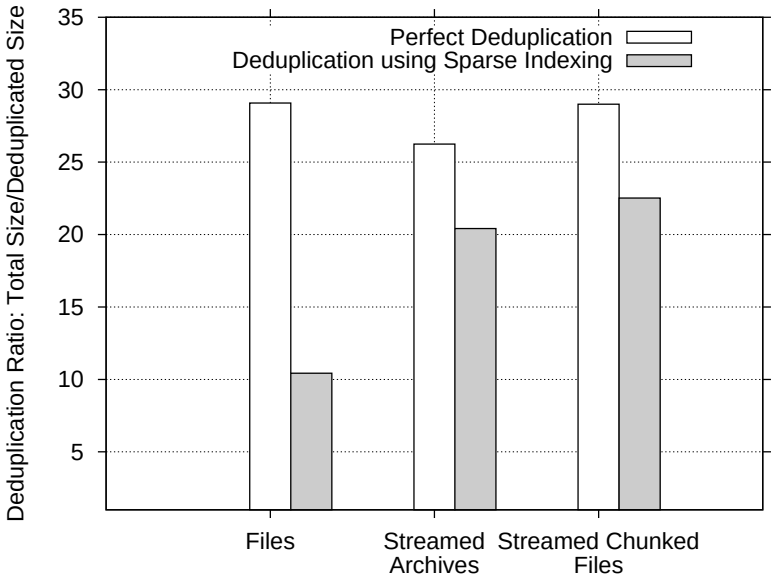

Figure 4.7: Deduplication for SE3D using min-hash sampling for Files, Streamed Archives, Streamed Files

deduplication is achieved when a full chunk index is maintained and all the duplicate chunks are removed. Due to the use of sampling, Sparse Indexing can miss some duplicates. It has been shown that this loss of duplicates is small [49] for traditional workloads.

Figure 4.6 shows that Sparse Indexing yields close to perfect deduplication ratios for the Streamed Archives and Streamed Chunked Files workloads, but not for the Files workload. The reason is that Streamed Archives and Streamed Chunked Files workloads contain high locality and hence, even with sampling, duplicates can be identified with high precision. The Files workload consists of small objects as compared to the 10 MB segments of the streamed workloads with little or no locality between objects. Due to this, fewer duplicates are identified. Hence, Sparse Indexing does poorly when deduplicating the Files workload. The figure also shows that better deduplication is obtained for the Streamed Chunked Files workload—16% less storage space was consumed over the Streamed Archives workload. This means that obeying file boundaries and keeping metadata chunks separate from content chunks has definite advantages. These observations hold for the SE3D data set as well as can be seen in Figure 4.7.

4.6.2 Deduplicating across workloads

Figures 4.8 and 4.9 shows Sparse Indexing's performance when deduplicating combination workloads for the Workgroup and SE3D data sets while using some variations of min-hash sampling. Figure 4.8 shows data points for the Workgroup data set for two combination workloads. The 'Streamed archives followed by files' is made up of the entire Streamed Archives workload followed by the Files workload. After the entire Streamed Archives workload, the same data is repeated all over again, except packaged as individual files. Hence, there

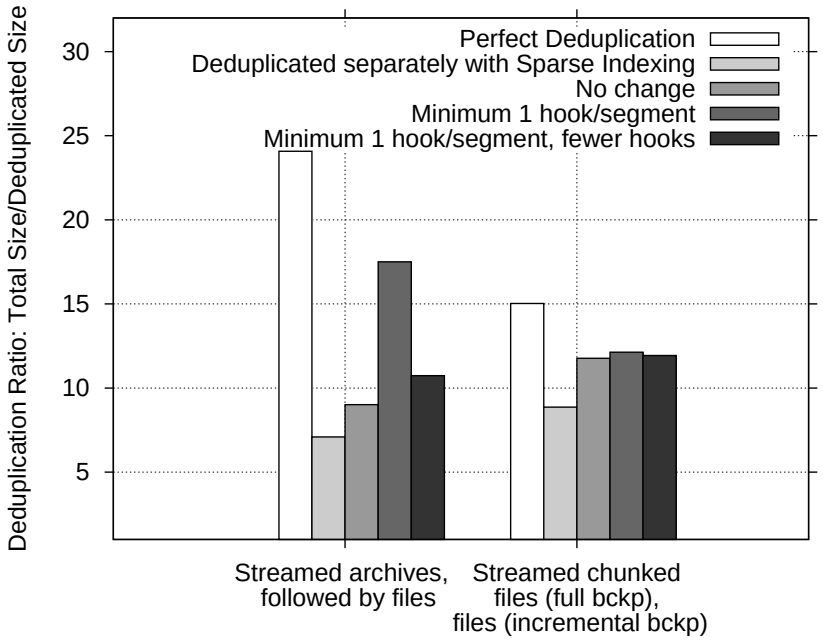

Figure 4.8: Deduplication for Workgroup using min-hash sampling for combination workloads

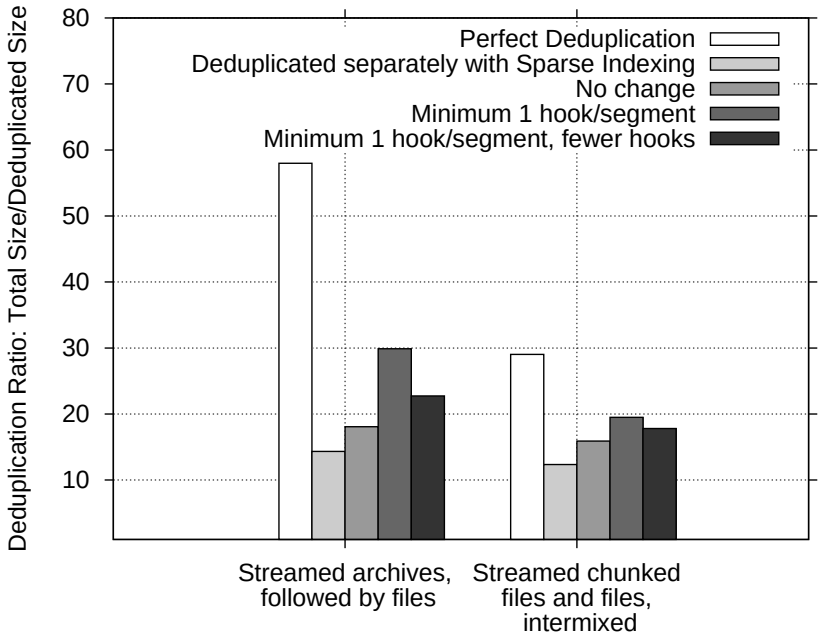

Figure 4.9: Deduplication for SE3D using min-hash sampling for combination workloads

is complete duplication of data as can be seen by the high perfect deduplication ratio of 24.07. This is compared with the deduplication ratio we would have achieved if the two workloads were deduplicated separately by dedicated systems. The storage space used by each of those systems was added to calculate the combined deduplication ratio. The figure shows that dedicated systems would yield a deduplication ratio of 7.09. However, if Sparse Indexing was used to deduplicate the *combined* workload, it yields better deduplication—a ratio of 9.01, a 21% gain in storage space savings over using dedicated systems.

By enforcing at least one hook per object, Sparse Indexing yields much better deduplication — a ratio of 17.50. In this case, a hook was extracted for every object, irrespective of its size. However, since more hooks were extracted, the number of hooks in the sparse index increased two fold. This means a proportionate increase of RAM usage too. To curb RAM usage, another modification was made. At least one hook was extracted per object. However, the Sparse Index was updated with only those hooks that would have been extracted anyways, even if this condition was not enforced. Thus, for objects that were too small to yield hooks, one hook was extracted to look up the index and deduplicate them, but this hook was not added to the Sparse Index. By doing this, we avoid having to invest RAM for objects that are too small. This technique yielded a deduplication ratio of 10.73, a 16% increase in storage space savings over the original scheme where hooks were extracted only if the object size was large enough to warrant it. The advantage of this scheme is that it has the same RAM requirements to that of the original scheme. Hence, this 16% increase in storage space savings comes at no additional RAM cost. Of course, additional I/O's will be required to deduplicate some small objects.

The same trend was observed for the second Workgroup combination workload, that was the 'Streamed chunked files (full bckp), files (incremental bckp)'. Here, streamed chunked files were used for the weekly full backup, and the incremental backup was file-based. This combination represents a case where a streamed workload was generated for a full backup so as to exploit the locality, but only modified files were sent one at a time during the week for incremental backup.

Combination workloads were also generated for the SE3D data set. The first workload 'Streamed archives followed by files' was generated similar to that of the Workgroup data set. The second workload 'Streamed chunked files and files intermixed' consisted of a mixture—some directory versions were packaged as streamed chunked files and some were packaged as individual files. The observations regarding the Sparse Indexing's performance are the same as for the Workgroup workload as shown in figure 4.9.

These results depicted in figures 4.8 and 4.9 show that Sparse Indexing when used as a unified deduplication engine for combination workloads yields better storage space savings compared to those obtained when using dedicated systems. By using min-hash sampling and its modifications, it is possible to further improve these savings.

4.6.3 Comparison of sampling methods

Figures 4.10 and 4.11 compare the two sampling methods: prefix-hash and min-hash for the Workgroup and SE3D data set respectively. We can see that for both data sets min-hash yields better deduplication than prefix-hash. The reason being that with min-hash every object always yields the predetermined number of hooks. This may not always be true for prefix-

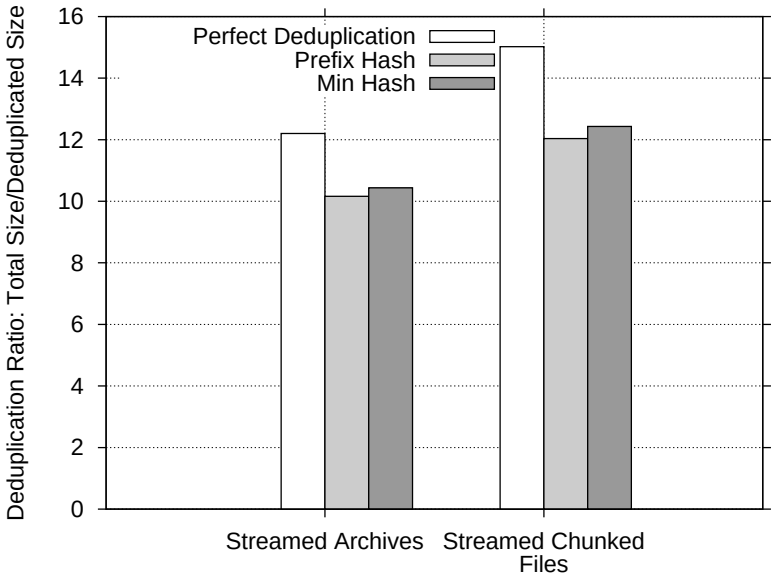

Figure 4.10: Deduplication for Workgroup when using prefix-hash and min-hash sampling

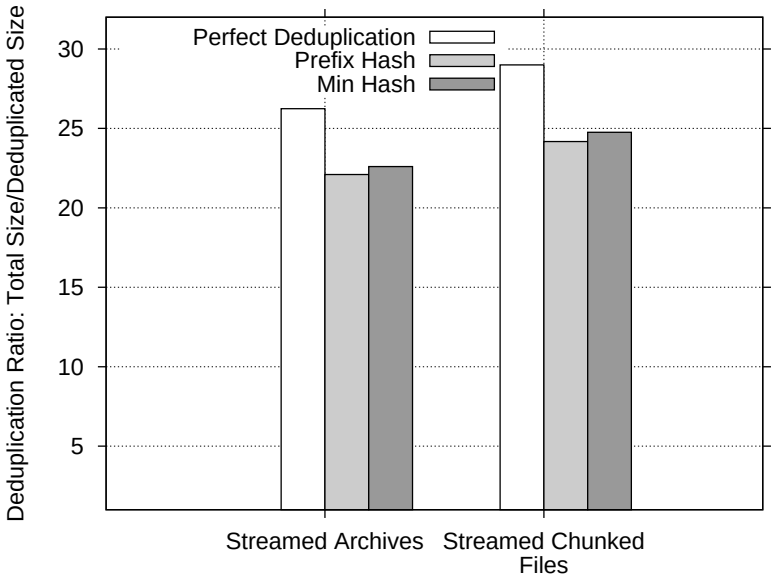

Figure 4.11: Deduplication for SE3D when using prefix-hash and min-hash sampling

hash. Also, using min-hash yielded only 1 to 2% more hooks than with prefix-hash. Therefore, the RAM overhead was low. It is also important to note that min-hash has other advantages over prefix-hash. It is more adaptable, in that the variations of minimum hooks per object, mentioned in the previous section can be implemented. This is not possible with prefix-hash. The disadvantage may be that some min-hash hooks may go obsolete in the long run. Some of the hooks may no longer be hooks as objects change over time and hence may not yield any benefits with respect to deduplication. A housekeeping process that cleans up such aged hooks may be implemented to phase out such hooks. However, this hook aging needs to be investigated in more detail.

4.7 Summary

Traditional backup workloads and file-based backup workloads have very different charateristics. The former is a long stream of bytes containing high locality; the latter consists of files of varying sizes coming in from disparate sources with little or no locality between files arriving within a given window of time. Sparse Indexing has been designed for a traditional backup workload. It exploits the inherent locality within traditional workloads to deduplicate very well while preserving throughput. Extreme Binning has been designed for file-based low-locality workloads. However, neither can deduplicate the other's workload while preserving their performance.

Extreme Binning cannot deduplicate large streams very well. However, we found that Sparse Indexing with min-hash sampling works well as a unified deduplication engine. Min-hash sampling is more adaptable than prefix-hash. It is ideal for workloads that contain

small objects because we can enforce a minimum hook per object condition. Real world data sets were used for our experiments. We found that obeying file boundaries when chunking and keeping metadata chunks separate from content yields better deduplication ratios.

Using these various techniques, we can build an adaptable unified deduplication engine based on the Sparse Index design. Such an engine can deduplicate high locality stream-based workloads very well, though it may not do as well in the case of low-locality small object workloads. However, our experiments have shows that combination workloads, where data is repeated but in different packaging, can be deduplicated by such a unified deduplication engine. Since duplicates are found *across* workloads generated by varied clients, the deduplication is better than that obtained when dedicated systems are used. Such a system is also easier to manage and administer than several dedicated systems deployed for each application/workload.

Chapter 5

Selective Chunk Replication for High Reliability

5.1 Introduction

Archival digital data continues to accumulate at an astounding pace. It will increase ten-fold between 2006 and 2010 to over 27 exabytes in the commercial and government sectors [55]. As digital data accrues at ever-increasing rates, organizations also face increasing regulatory pressure to retain data for long periods of times and may be required to retrieve data occasionally. In this context, maintaining the availability of archived data becomes part of the due diligence that organizations are expected to exercise.

To reduce the costs incurred for storing such large volumes of archival data, this data is compressed using various compression techniques. Several companies [27, 28, 33] already use various forms of compression for their archival storage solutions. Our project, Deep Store [84], uses both intra-file and inter-file compression to reduce redundancies. One such

inter-file compression technique used by Deep Store is chunk-based inter-file compression [57]. In this technique files are split into variable-length chunks and stored. If any redundant chunks are found, they are stored as references rather than as duplicates. In many cases, this method achieves excellent compression ratios [83].

While archival systems require good compression, they must also ensure that data is preserved over long time periods. Compression techniques, while they save storage space, also have the potential to reduce reliability. For example, when inter-file compression is used, dependencies are introduced between files that share the same chunk. If such a shared chunk is lost, a disproportionately large amount of data becomes inaccessible because of the loss of all the files that share this chunk. As a result, some chunks are much more important than others and need to be protected at a higher level to maintain good overall reliability. Here, we consider the effects of inter-file chunk-based compression on the reliability of the archival system. Our approach to improving reliability is to add redundancy strategically by *selectively* replicating chunks. We have developed heuristics that weigh the importance of a chunk and use this weight to prescribe the level of replication for the chunk. A part of the storage space saved by compression is thus reinvested in better protecting the important chunks. As a result, we achieve even better data reliability than mirrored (degree of mirroring = 2) Lempel-Ziv (LZ) compressed [88] files, while still using about half of the storage space of mirrored LZ-compressed files and with replication/mirroring as the means to introduce redundancies.

We can also improve reliability by using other redundancy introducing techniques such as erasure correcting codes used in RAID levels 5 and 6, and by introducing different data placement, failure detection and recovery disciplines. We do not consider these here mainly

because they offer intricate trade-offs between speed of recovery, ease of recovery, and computational and storage overhead.

To focus our efforts, our analysis assumes constant device failure rates, constant repair rates, and independence of failures. We only investigated replication as a redundancy strategy and used a simple concept of *robustness*, in which we measured the amount of data loss caused by the loss of a small percentage of devices in addition to a standard failure model with all usually made simplifying assumptions. We will investigate other redundancy strategies in the future.

5.2 Deep Store: An Overview

Deep Store [84] is a large-scale archival storage system that stores volumes of immutable data efficiently, with high reliability and accessibility. It incorporates methods for inter-file and intra-file compression to utilize storage space very efficiently. Deep Store uses three techniques to reduce storage demands: content-addressable storage [33], delta compression [4, 29] and sub-file chunk-based compression [57]. Other storage systems such as Venti [66], EMC Centera [33], StorageTek's Intellistore [77], Nexsan's SATABeast [60], Avamar's Axion [8], and Permabit [64] also use content addressable storage. In content-addressable storage, a single feature or hash, also called *content address* is computed over an entire file and this hash is used to find identical files already in the archive. Two files with the same content address are likely to be identical, but the system still must check for possible collisions. If the files are identical, the system only stores a reference to the existing file rather than storing the file again. In delta compression, the system first searches for a file similar to the file currently being stored, and

then stores only the differences between the current file and the stored file. A pointer to the stored file and metadata for reconstructing the current file are stored with the differences.

Our study focuses only on chunk-based compression. Chunk-based compression or *chunking* subdivides a file deterministically into variable-sized blocks or chunks. This technique was first used in the Low-bandwidth Network File System [57]. Chunking is a two step process. First, a file is divided into chunks in a deterministic fashion. Second, the content within every chunk is used to compute its features. This process of chunking has been described in section 3.2. You *et al.* [83] have evaluated chunking and delta-compression with respect to their storage space efficiency and computational complexity. They conclude that delta-compression and chunking outperform traditional stream compression methods with respect to storage space efficiency. Chunking requires two hashing operations per byte in the input file: one fingerprint calculation of the fixed size window and one chunk digest calculation. Once the file is broken into chunks, only the unique chunks are actually stored. Deep Store identifies a chunk in the same way as it identifies files: using a content address (a hash or digest of the content) to determine if a chunk already exists in the system. After this type of compression, a file consists of a set of references to chunks and the metadata necessary to rebuild the file.

5.3 Effect of Compression on Reliability

Chunk-based interfile compression can be quite effective for certain types of data. You, *et al.* [83] have characterized this data as files that evolve slowly mainly through small changes, additions, and deletions. One of the data sets for our experiments consists of 9.8 GB of several web sites: those of the University of California at Santa Cruz, Santa Clara University,

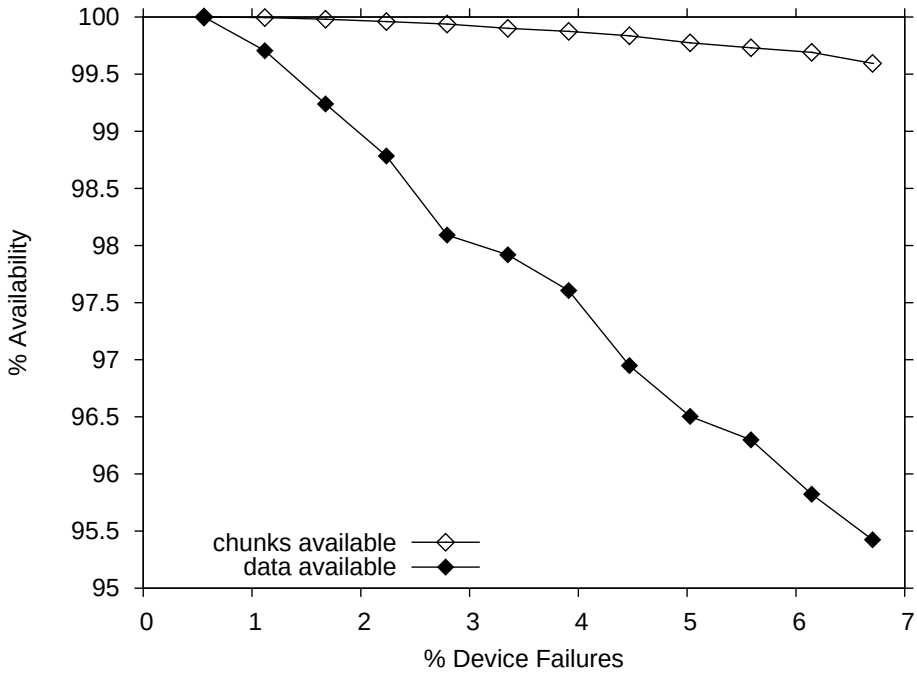

Figure 5.1: Effect of inter-file dependencies on robustness

Stanford University, University of California at Berkeley, BBC, NASDAQ, CERT, CNN, SANS, SUN, CISCO, and IBM as they developed over time. We obtained them from the Internet Archive's Wayback machine [78]. This data is a representative sample of archival data, and will greatly profit from chunk-based compression due to the incremental nature of the changes that it has gone through. Chunk-based inter-file compression stores this data using a storage space of 1.74 GB for chunks and 280 MB for metadata. On the other hand, when each file was compressed using LZ-compression, the total storage space required was 5.6 GB. Clearly, chunk-based compression can use significantly less storage space than LZ-compression.

To study the effect of chunk-based compression on reliability we conducted a pilot experiment using this data. We compressed the files using chunk-based compression, and then mirrored the chunks and stored them evenly across a set of 179 devices. The devices were then randomly selected to fail independently, resulting in the loss of up to 7% of the total devices. Figure 5.1 shows the availability as a function of device failures in two forms. The first form is the percentage of raw chunks available. The second form is the percentage of original data that could be reconstructed from these available chunks. The data robustness is seen to be significantly lower than the chunk robustness. For example, when 6% of the devices fail, about 99.5% of all chunks are still available, but only 96% of all the data is still available. This increased data loss happens due to inter-file dependencies formed as common chunks are shared amongst multiple files. These inter-file dependencies are shown schematically in Figure 5.2 where the dependencies are measured by the number of file references to a chunk. If a common chunk is no longer available, all the files that depend on the chunk are lost resulting in a disproportionately large amount of data loss that we see in Figure 5.1. This increased data loss illustrates

how good compression can be detrimental to reliability in the event of device failures, due to inter-file dependencies formed by common chunks shared between multiple files.

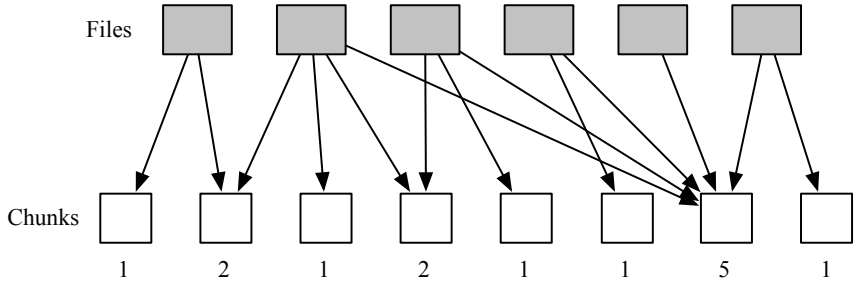

Figure 5.2: Inter-file dependencies

Chunk-based compression achieved excellent compression ratios by removing redundancies across files. However, this introduced inter-file dependencies that hampered reliability. Since compression saves significant amount of storage space, some of this savings can be used to regain reliability. A simple way of doing this is to use a higher degree of replication. However, we have used a more discerning approach to do this — one that decides the replication level for a chunk depending on its popularity or importance so that we did not end up defeating our original purpose of efficient storage space utilization.

5.4 Storage Strategy

The simple experiment in the previous section showed that the loss of a small number of chunks can result in a disproportionately large data loss. To protect against this, our heuristics replicate certain important or popular chunks more aggressively than the others. To

accomplish this, we developed some good measures for the importance of a chunk. This measure of importance, or *weight*, is used to determine the number of replicas for each chunk and their distribution across devices.

5.4.1 Replicas Based on Chunk Weight

The effects of the loss of a chunk can be measured by the amount of data lost and by the number of files that are inaccessible as a result of this loss. Correspondingly, we measure the importance of a chunk either by the number of files that depend on it (the reference count), or by the amount of data (the byte count) that depends on it. This approach defines the weight of a chunk as either the reference count or as the byte count that depend on it, and determines the number of replicas for each chunk using a logarithmic function of the chunk's weight.

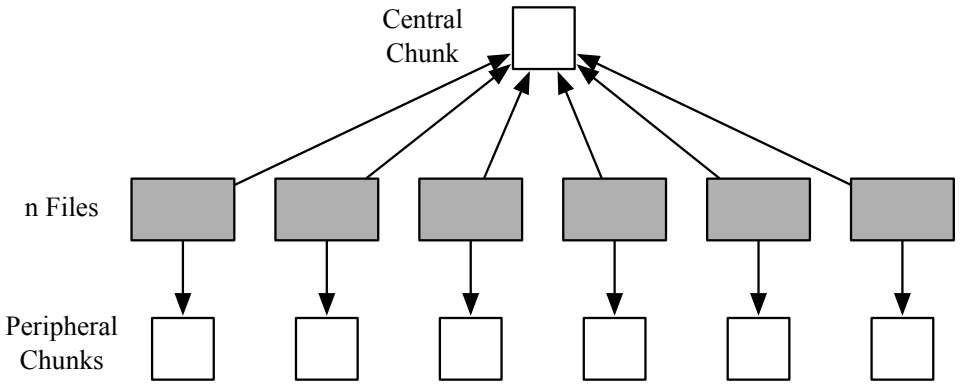

Figure 5.3: Central and peripheral chunks

The following calculation justifies our intuition to use a log-based function to calculate the number of replicas for a chunk based on its weight. Assume that we have n files that

all depend on one common chunk, called the central chunk. Each file also depends on another peripheral chunk, as shown in Figure 5.3, that is particular to that file alone. Assume that we keep k replicas of the single central chunk and l replicas of the remaining peripheral chunks. We assume that all chunks have the same size. The total storage used is then proportional to

$$S = k + l \cdot n$$

Assume that a single storage device fails with probability p. The central chunk is lost with probability p^k and the peripheral chunks with probability p^l each. We lose all files if we lose the central chunk; otherwise the expected number of lost files L is np^l, so that

$$L = np^k + (1 - p^k)np^l \approx np^k + np^l$$

Taking the derivative of the approximation for the loss, we obtain the following relation for an optimal k:

$$n \log(p)p^k - \log(p)p^l = 0$$

Solving for k gives

$$k = \frac{S}{n+1} + \frac{n}{n+1} \cdot \frac{\log(n)}{\log(1/p)}$$

The first addend converges to zero and the second is proportional to the $\log(n)$. If the central chunk is much larger than the smaller chunks and n is fixed, then replicating the peripheral chunks at a higher rate than the central chunk leads to lower expected loss.

Even if a chunk has only a single dependency, it must be protected. Therefore, our heuristic keeps at least 2 copies of every chunk. We choose a function of type

$$k = f(w) = \min(\max(2, a + b \log(w)), k_{max})$$

to calculate the number of replicas k depending on a chunk's weight w. Here, a and b are constants that will yield different storage space utilization and robustness levels; a and b need to be determined experimentally depending on the data set. A base level of replication, a, is added as an additional tuning parameter that is independent of w to offset the effect of $b\log(w)$. As b increases, the number of replicas, based on the weight w of a chunk, increases. For some chunks with a large weight, w, the number of replicas suggested by our logarithmic function can be very large. As k increases the gain in reliability obtained due to each additional replica diminishes. For this reason, the maximum number of copies of a chunk is capped at k_{max}.

5.4.2 Chunk Distribution

In addition to the replication level for various chunks, the placement of the replicas also affects the reliability of our storage scheme. If a device is lost and almost all chunks on the device belong to the same set of files that reside on the lost device, then the effect of this failure has limited repercussions for the rest of the system. Conversely, if a file depends on chunks distributed over a large set of devices, then it is more vulnerable since it is easier to lose this file through the failure of any of those devices. Consequentially, we want to reduce *inter-device* dependencies. Of course, we should store copies of the same chunk on different devices. Other than that, we try to store chunks belonging to the same file on the same device.

Since our system stores archival data, we assume that files enter the system in batches. As a file enters, the chunks are extracted and stored, filling up the disks as data arrives, on one disk at a time. When the current disk is full, a new disk is used. If a chunk is new, it is stored on the current disk, but not yet replicated in anticipation of another file in the same batch using

the same chunk. This lazy replication scheme reduces inter-device dependencies. If a chunk is already in the system, the system determines whether, after updating its weight, another replica must be stored. If this is the case, the replica is stored on the current disk. Otherwise, the system does nothing—there are sufficient replicas for the chunk already. After the batch of files has finished processing, the weights of all the chunks are checked to see if any of them need to be replicated. In such cases, replicas for the latest chunks are stored on the most recently used disk. While our scheme does not completely eliminate inter-device dependencies, it greatly reduces them.

5.5 Experimental Setup

Our data set consists of two sets of files obtained from the Internet Archive [78] and the other from the *Santa Cruz Sentinel* [79]. As described in Section 5.3, the data set from the Internet Archive contains web sites as they develop over time. *The Santa Cruz Sentinel*, our local newspaper, maintains an archive, as do many newspapers. This set consists of HTML, PDF, image (TIFF and JPG) and Microsoft Word files with quite a bit of repetitive information such as templates for web pages. Table 5.1 gives statistics for both data sets, showing that both data sets are well-suited for chunk-based compression. The use of chunk-based compression results in substantial savings in storage space when compared to the storage space required when using LZ-compression to compress each file individually.

We used our prototype program `chc` [84] to chunk files. The files that form the target data set were input to `chc`, producing an output composed of chunks derived from the original files. These chunks were further compressed individually using the *zlib* [32] compression

Table 5.1: Statistics of the Experimental Data

	Internet Archive	Santa Cruz Sentinel
Number of Files	196664	158900
Minimum File Size	1 B	2 B
Maximum File Size	21 MB	263.78 MB
Average File Size	52.50 kB	301.46 kB
Total File Space	9.84 GB	40.22 GB
LZ-compressed File Space	5.62 GB	31.14 GB
Unique Chunks	6240360	28806477
Minimum Chunk Size	9 B	9 B
Maximum Chunk Size	12.61 kB	12.61 kB
Average Chunk Size	299.90 B	243.11 B
Total Chunk Space	1.83 GB	7.5 GB

library. `chc` captures a list of chunk identifiers for each file, as well as the identifier and size for each chunk. Extended size blocks—*megablocks* [84]—were used to store both chunks and LZ-compressed files to minimize the storage overhead from unused portions of blocks. Since the metadata for file identifiers and size of every file needs to be stored for LZ-compressed files as well, the storage overhead due to this metadata has been omitted for both chunk-based compression and LZ-compressed files. However, the overhead of a 128-bit content address for every chunk, whether original or replica, and for all chunk identifiers per file has been accounted for when calculating the total storage space required when using chunk-based compression.

To evaluate the success of our heuristic-based replication strategy, we measured the ratio of availability to the utilized storage space. Evaluation of the latter is easy, while the former is difficult because availability depends on too many factors such as data placement, speed of recovery and device failure rates. Further, availability calculations make simplifying assumptions that are not always justified, such as independent failures of devices and constant device failure rates. In addition, an archival storage system tries to protect data over a period of time that is longer than the lifespan of the individual devices and, in such a system, common causes of failures such as batch and vintage failures become important. Instead of trying to make a number of reasonable assumptions and ending up with a large number of possible storage systems, we decided to measure availability in the form of *robustness*, defined as the fraction of data available given a certain percentage of unavailable storage devices, rather than in the usual metric of mean time to data loss or percentage of data loss per year.

In this assessment, we assume a simple model based on replication—the only way we introduce redundancy is by storing more replicas. Though we decided to use replication instead

of more involved mechanisms to generate redundancy, there are still many potential parameters to choose in a storage system, such as replica placement, failure detection and repair. Since our target applications are so large that they store data on hundreds, if not thousands of disk drives, we use artificially small devices to store the data so that our sample workloads are stored over many devices. By not modeling repairs of failed devices, we are being conservative. This is important because, in any real system, repairs would occur after a failure, so there would be a much smaller chance of data loss.

To test the robustness of the system, we began by selecting a percentage of devices independently and at random and failing them, starting with 1 device and continuing until 7% of the total devices have failed. By showing availability at relatively low levels of device failure, we simulated the effects of temporary device loss. The devices would be replaced later, but the data on them is lost due to failure. The same is true for mirrored LZ-compressed files. We used the chunk distribution strategy of Section 5.4 to store chunks extracted from both the data sets onto a set of devices. The same distribution strategy was used with LZ-compressed files. The mirrored LZ-compressed files of the Internet Archive were stored evenly on 188 devices of 64 MB each while those of the *Santa Cruz Sentinel* were stored evenly on 132 devices of 512 MB each. However, every device was filled to capacity. Hence, measuring the *percentage* of failed devices was equivalent to measuring the percentage of data lost. The capacity of every device was increased for *Santa Cruz Sentinel* data to avoid fragmentation of a file across several devices. We had to take care not to fragment files when using LZ-compression because this would introduce the same type of multiple-device dependencies for files that arise when using the chunking method, the effects of which we were measuring. Chunks for both data sets were

stored on smaller devices than those used when storing the same data that was LZ-compressed to make sure that we distributed chunks onto the same number of devices as those used by the mirrored LZ-compressed files thereby facilitating a fair comparison between the two. We ended up using an additional 5 disks on average when storing chunks. We could not ensure using *exactly* 188/132 devices since it was not possible to know apriori the number of redundant chunks that would be added with different redundancy schemes. Once we randomly chose the failed devices, we then calculated how many files and how much data we could reconstruct using the remaining devices. The performance of chunk-based compression was compared with that of LZ-compressed files on the basis of the robustness and storage space consumed.

5.6 Results

We calculate the weight, w, of a chunk using two heuristics: the number of files and the size of data depending on a chunk. The weight of a chunk in terms of the number of files, F, depending on a chunk, is calculated as $w = F$. The weight of a chunk in terms of the size of the dependent data is calculated as $w = D/d$, where D is the sum of the sizes of all the files that depend on this chunk and d is the average size of a chunk. The number of replicas, k, calculated using w in the log based function of Section 5.4.1, is rounded off to the nearest integer. For each experiment we have measured the storage space used as a percentage of the storage space used by the original *uncompressed* data.

The first set of experiments demonstrates the use of the two heuristics. We wanted to study how the robustness is affected by varying the base level of replication, a. By increasing the base replication level, the number of replicas for *all* the chunks increases, resulting in better

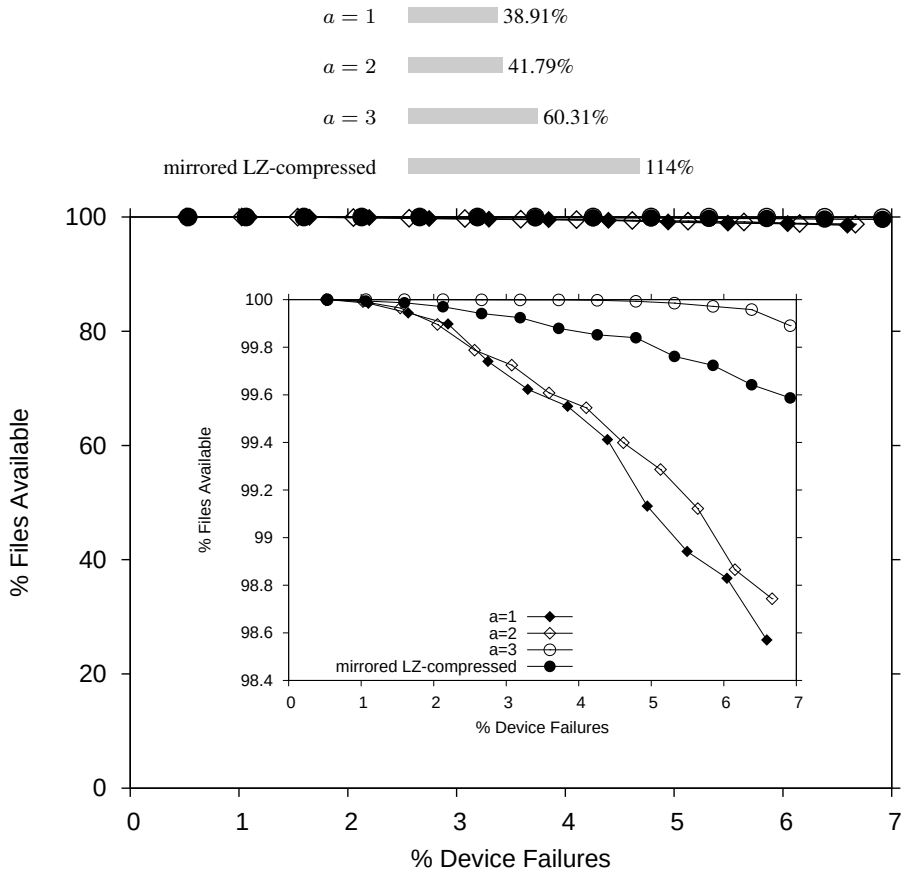

Figure 5.4: Effect of a on robustness using heuristic $w = F$ with $b = 1$, $k_{max} = 4$

robustness. The results of these experiments, conducted using Internet Archive data and with $k_{max} = 4$, are shown in Figures 5.4 and 5.5.

In Figure 5.4 we show robustness using the number of files depending on a chunk as a heuristic, *i. e.* $w = F$. The inset shows the same graph with an expanded Y-axis to show the small differences. Hence, we measured availability in the number of files available, not amount of data available. Here, with $b = 1$, we vary a and see that the robustness increases with increasing values of a. The system is not very robust when $a = 1$ because when using $a = 1$, 90% of the chunks were replicated just once. We showed in Section 5.3 that when all the chunks are uniformly replicated just once, the robustness suffers. We see the same effects when $a = 2$, where around 80% of the total chunks were replicated just once. At $a = 3$, our system is *more* robust than mirrored LZ-compressed files and uses only 52.75% of the storage space required by mirrored LZ-compressed files.

In Figure 5.5, we show a similar effect of a on the robustness, but, using dependent data as a heuristic, *i. e.*, $w = D/d$, with $b = 0.4$. Again, we see that by increasing a the system's robustness improves. At $a = 0.5$, our system is more robust than mirrored LZ-compressed files and uses only 61.20% of the storage space used by mirrored LZ-compressed files. Further increase in a increases the robustness even more, albeit at the expense of additional storage space.

The results of the above experiments show that the robustness of our system exhibits the same trends when we use either heuristic, $w = F$ or $w = D/d$. The rest of the results presented here use dependent data as the heuristic, *i. e.*, $w = D/d$; however, the same trends are found with the number of references being used as a heuristic.

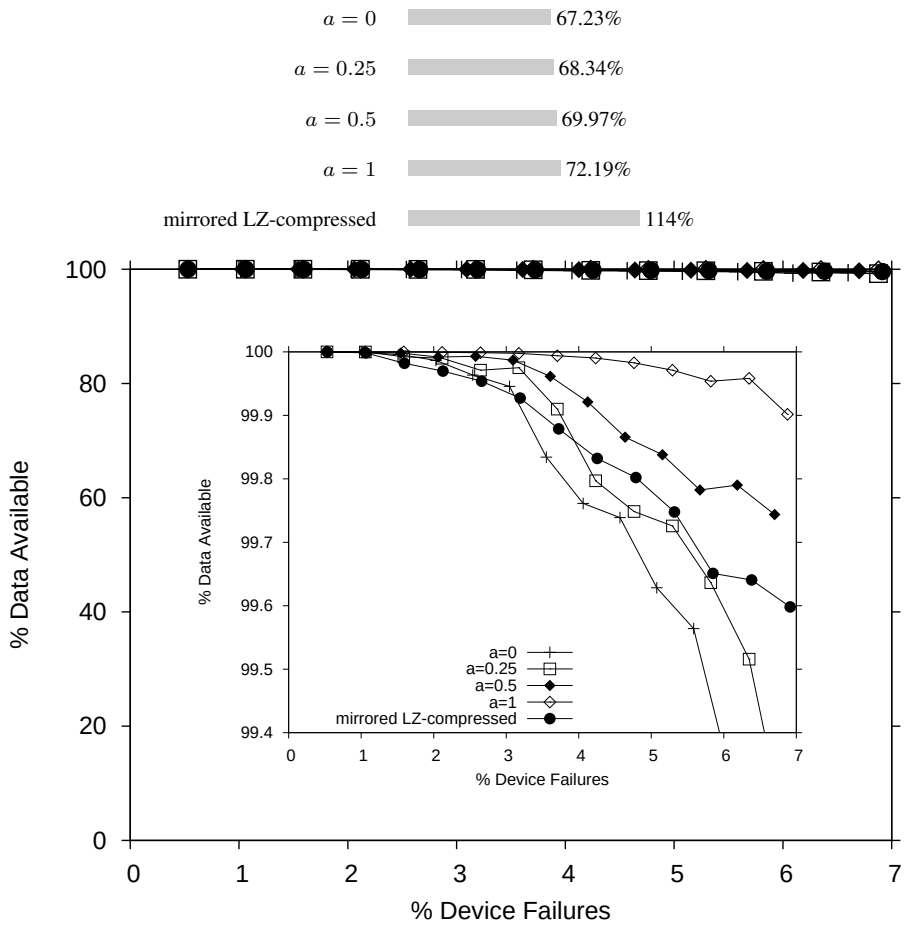

Figure 5.5: Effect of a on robustness using heuristic $w = D/d$ with $b = 0.4$, $k_{max} = 4$

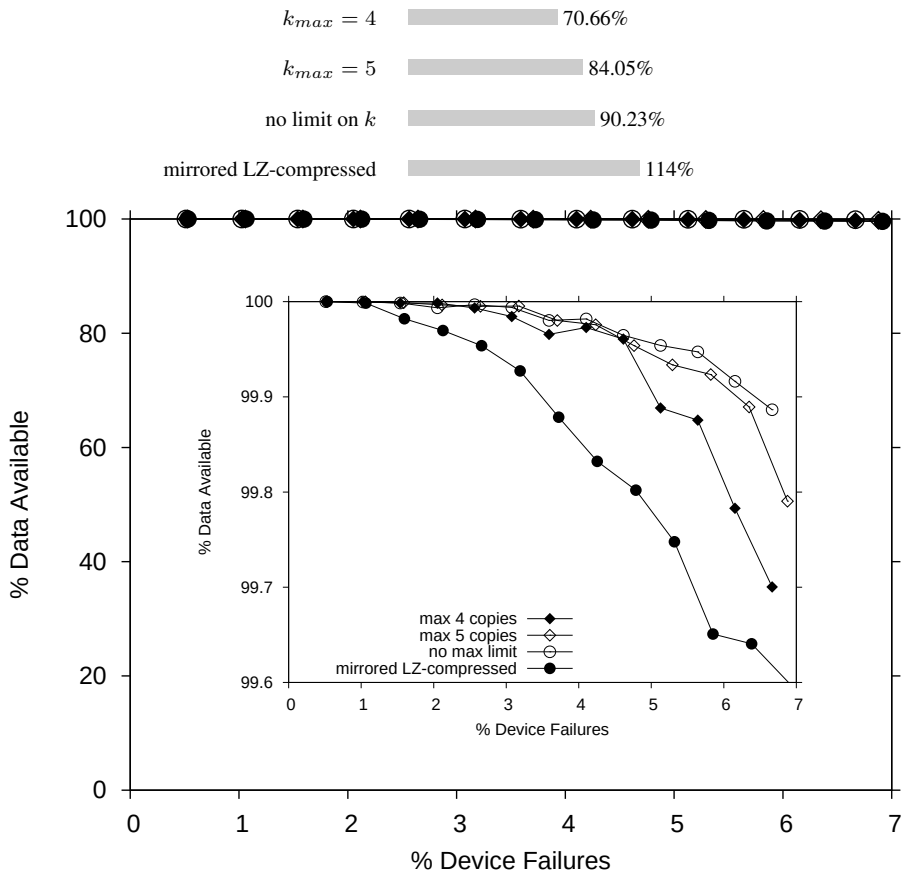

Figure 5.6: Effect of limiting k on robustness using heuristic $w = D/d$, $b = 0.55$, $a = 0$

If we do not restrict the number of replicas of a chunk, k, to a predefined maximum, k_{max}, some chunks end up having a very large number of replicas, especially for higher values of b. However, as the number of replicas increases, the gain in robustness that every replica rewards us with diminishes in value. To study the effect of varying k_{max}, we measured the robustness of the Internet Archive data with $b = 0.55$ and $a = 0$, as shown in Figure 5.6 for $k_{max} = 4$ and $k_{max} = 5$ compared with that obtained with no limit on k. It is clear that limiting the number of replicas with k_{max} does not result in a noticeable loss in robustness, but *does* result in significant savings in storage space.

In our next experiment, we studied the effects of varying b, which will improve robustness by increasing the number of replicas for the more important (higher weight) chunks. This comparison is shown in Figure 5.7 using Internet Archive data. As b increases, for a given w we begin to get higher values for k, resulting in an increase in the storage space required and the robustness of the system as can be seen in Figure 5.7. At $b = 0.55$, our system is more robust than mirrored LZ-compressed files, but uses only 61.98% of the storage space required by LZ-compressed files.

Figure 5.8 depicts the robustness of the second data set, from the Santa Cruz Sentinel, when using different values for b. Here, too, our approach is more robust than when using mirrored LZ-compressed files. With $b = 1$, we use only 48.41% of storage space of the base LZ-compression approach.

As we increase the redundancies the storage space required by metadata also increases. For the Internet Archive data the storage space used by the metadata constituted 5%

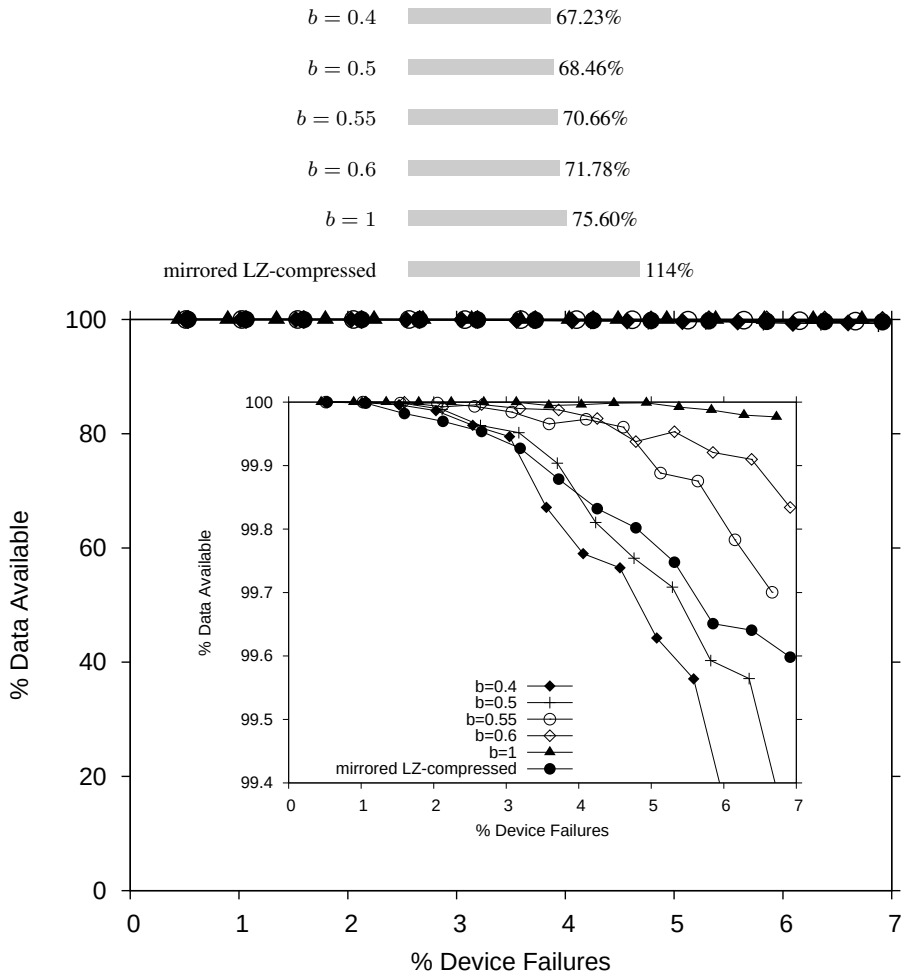

Figure 5.7: Effect of b on robustness using heuristic $w = D/d$ with $a = 0$, $k_{max} = 4$

of the total storage space. For the *Santa Cruz Sentinel* the metadata required 5.6% of the total storage space.

We have used both, the number of files and the amount of the dependent data as heuristics for determining the weight of a chunk. The choice of heuristic depends on the corpus. If the sizes of files in a corpus are indicative of their importance, then the dependent data heuristic should be chosen. However, if the importance of a file in a corpus is independent of its size, or all the files in the corpus are equally important, then the number of files should be chosen as a heuristic. The same metric used in the heuristic must then be used for measuring the robustness of the system; *i. e.*, when using the number of files in the heuristic we use the number of available files as the measure of robustness, whereas when using dependent data as heuristic we use the amount of available data. In other words, if all the files are equally important, then one should measure the system robustness in the number or percentage of files available. We have investigated the effects of the parameters a, b and k_{max} on the robustness and the storage costs of an archival system using chunk-based compression. By choosing an appropriate combination of these parameters we can achieve both a higher robustness and lower storage space utilization compared to traditional LZ-compression techniques.

5.7 Related Work

Several systems that exploit data redundancy at different levels of granularity have been developed in order to improve storage space efficiency. One class of systems detects redundant chunks of data at granularities that range from entire file, as in EMC's Centera [33],

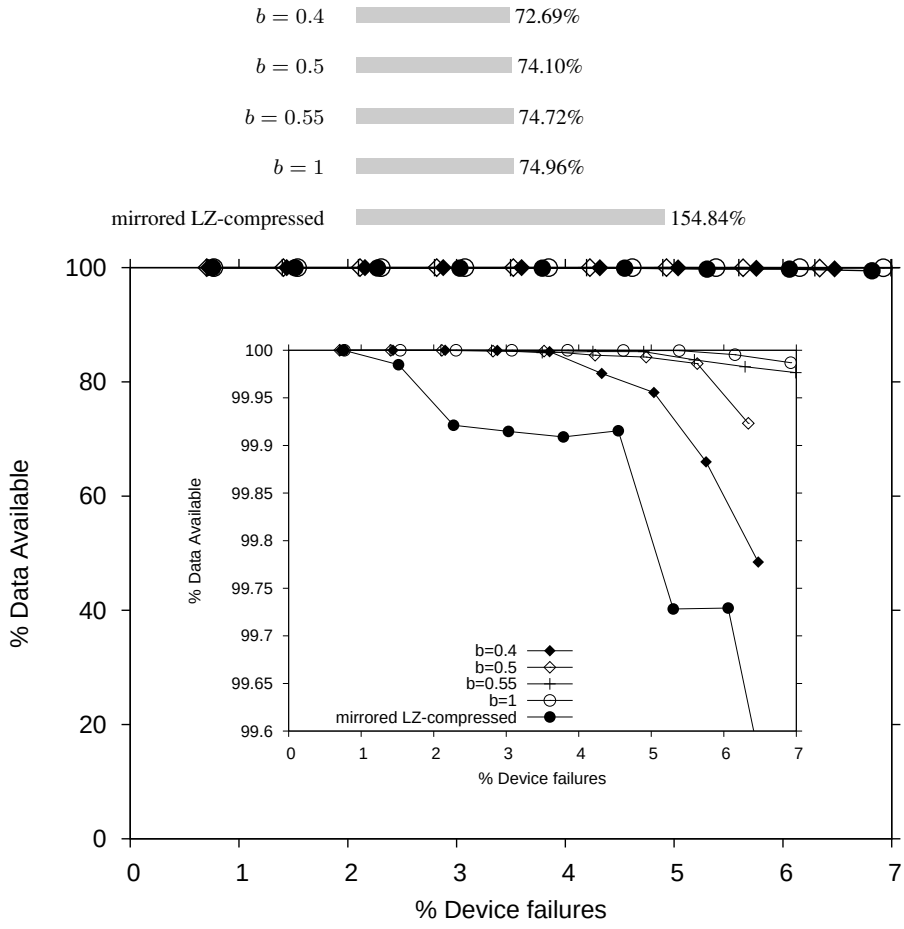

Figure 5.8: Effect of b on robustness using heuristic $w = D/d$ with $a = 0$, $k_{max} = 5$, Santa Cruz Sentinel data

down to individual fixed-size disk blocks, as in Venti [66] and variable-size data chunks as in LBFS [57].

RAID [21] is a device driven method for introducing redundancy and thus ensuring the reliability for storage systems. OceanStore [46] aims to provide continuous access to persistent data on a global scale and uses automatic replication strategies to boost reliability of the system in the face of disasters. FARSITE [3] is a distributed file system that achieves reliability through replication of file system metadata, such as directories, and file data. FARSITE chooses replication instead of erasure coding schemes to avoid the additional overhead of latter when reconstructing a piece of information. Other file systems such as PASIS [39] and Glacier [40] also make use of aggressive replication to guard against data loss. The LOCKSS project [53] uses a peer-to-peer audit and repair protocol to preserve the integrity and long-term access to collections of documents. Baker *et al.* [9] suggest that long term reliability additionally requires auditing the integrity of data above the level of the storage devices. The surplus storage space we save by using interfile compression can be used to implement proactive policies for ensuring reliability [82], verifying the data integrity [75], and developing recovery strategies [81] for large scale storage systems.

5.8 Summary

The chunk-based inter-file compression yields very good compression ratios by removing inter-file redundancies. However, these reduced redundancies can be detrimental to the robustness of the data. We have presented a simple strategy to increase the robustness of data with chunk-based compression without compromising the storage space savings obtained

by the compression. Our strategy allows us to control the balance between storage space savings and reliability by a choice of heuristics and parameter variation. This strategy gives both a higher robustness and significant storage space savings compared with traditional LZ-based compression.

We have shown that choosing the right number of replicas for each data chunk can achieve a much higher robustness while using about half of the storage space required by mirrored LZ-compression. Furthermore, by controlling the parameters in our replication strategy, we can achieve an even higher robustness (close to 100%) for a small percentage of device failures. This higher robustness together with the savings in storage space is useful for future inclusion of repair models for chunk-based archival systems that use such inter-file compression. By adjusting the number of replicas of individual data chunks based on our heuristics, Deep Store and other long-term archives can reduce storage space requirements and thus costs while simultaneously increasing robustness, making the long-term storage of data both more affordable and more reliable.

Chapter 6

Conclusions and Future Directions

This dissertation concludes with a summary of our contributions along with their conclusions, and a discussion of possible future research in the area of scalable chunk-based deduplication.

6.1 Similarity-based Searches and Document Routing Algorithm

Finding similar documents within repositories using brute force methods, such as, pair-wise comparison is not a scalable solution. As repositories grow, the number of such comparisons grow quadratically making such a solution inefficient. To find highly similar documents quickly, each document is first processed to extract its set of features. The features are such that if two documents share even a single feature, then they share large stretches of overlapping content. Now, similar documents tend to share more features than dissimilar documents. Using the feature set of every document, we now use the measure of set similarity to find other similar documents efficiently.

Broder's theorem states that the probability that two documents have the same minimum hash is the same as their similarity measure. This means that if two documents are very similar, their minimum feature is the same with very high probability. Using this intuition, we developed a similarity-based search algorithm that uses only the top few feature hashes of a document to quickly find other highly similar documents in the repository. Feature hashes are held in an index for fast query. The feature index maintains a mapping between every feature and the list of files that this feature occurs in. To find similar documents to an incoming document only the top few feature hashes of the incoming document are used to query the index. The result is a set of documents in the repository that are highly similar to the incoming document.

Such an index can be partitioned to support the scale out of the repository. In this case, which partitions to query for any given document is dictated by the document's features themselves. Every document is routed based solely on its contents to only a small fraction of the total partitions. Those chosen partitions are queried using only the top few feature hashes to find documents that are highly similar to the incoming one. This can be done while still preserving the precision of the results—highly similar documents are found, less similar documents may or may not depending on their degree of similarity. The routing factor is a tunable parameter of the document routing algorithm. By varying it we can control how many partitions should be queried per document, and how many features should be used. Increasing the routing factor results in a larger result set—more similar documents, whereas a small routing factor may result in finding only the highly similar ones. In practise, we find that even with a scale out as large as 128 index partitions, only 3% of the partitions can be queried while still producing high

quality results. These results prove that this similarity-based document routing search solution is a scalable one.

Similarity-based searches in large scale repositories is only one of the applications for our document routing algorithm. Besides archival and backup systems our document routing algorithm can be used to distribute and locate content in peer-to-peer cooperative storage and backup systems [25, 30, 48] and distributed storage systems [3, 46]. Such systems can save storage space by routing documents to nodes that are expected to store similar content. The recipe for a document [80] consisting of the feature hashes can be used to locate it without having to consult a large number of indices.

6.2 Extreme Binning

Traditional workloads are large byte streams that contain a lot of locality. State-of-the-art techniques, such as, Sparse Indexing and Locality Sensitive Caching exploit this locality while deduplicating such workloads to alleviate the disk bottleneck. However, in the absence of locality neither approach can maintain its performance. File-based workloads are workloads where files are sent from disparate sources. No locality can be assumed between files arriving within a given window of time. For such workloads, we designed Extreme Binning. Extreme Binning relies on file similarity instead of locality. Deduplication requires only one disk access per file thus alleviating the disk bottleneck problem. Extreme Binning economizes RAM usage because it requires that only one entry per file be made in the RAM index.

Extreme Binning splits up the chunk index into two tiers. The top-tier sits in RAM as is called the primary index. The second tier is made up of many small indices called bins.

All the bins reside on disk. Every entry in the primary index points to one bin on disk. Bins are not shared between any primary index entries. After a file has been chunked, a list of the file's chunk IDs is generated. The minimum chunk ID is chosen as the file's representative chunk ID. This representative chunk ID is used to query the primary index and the corresponding bin is loaded from disk. The bin is queried for the rest of the file's chunk IDs. If no entry for the file's representative chunk ID is found in the primary index, a new entry along with a new bin is created.

A single entry is made in the primary index per file resulting in a low RAM footprint and only one disk access is required per file. Extreme Binning scales well without sacrificing deduplication quality. It scales out by allowing parallel file deduplication. Backup nodes can be added as when a scale out is required. First, the primary index is split up and distributed amongst the backup nodes. Next, the bins are distributed in tandem with their corresponding primary index entries. Due to bin independence, partitioning the two tier chunk index is an easy and clean operation. Since the data chunks are also not shared between bins, re-distributing the data is straightforward as well. No dependencies need to be resolved when moving indices or data.

In a distributed setting, with multiple backup nodes files are allocated to a single node for deduplication and storage using a stateless routing algorithm – meaning it is not necessary to know the contents of the backup nodes while making this decision. Maximum parallelization can be achieved due to the one file-one backup node distribution. Experimental results show that data gets distributed evenly among the various backup nodes. This shows that no single node will get overloaded. Even load distribution is important in distributed systems because

the performance of the system is dictated by the slowest node. The autonomy of backup nodes makes data management tasks such as garbage collection, integrity checks, and data restore requests efficient. The loss of deduplication is small and is easily compensated by the gains in RAM usage and scalability. Thus, Extreme Binning is an efficient, scalable solution for file-based backups.

Further analysis regarding the behavior and design of bins, and storage strategies for data chunks needs to be done. The behavior of bins, the relationship between the number of bins and the degree of duplication will help anticipate and design a robust system. For instance, we would like to know how bins grow, whether any bins get so big that fetching them into RAM would require more than one disk access and what fraction of the total number of bins is made up of such unwieldly bins? If there indeed exist such bins, how do we handle them? Should large bins be partitioned into smaller ones and if so, what is the best way to do it? Strategies for data placement are influenced by the fact that just as there is no sharing of chunk IDs between bins, there is no sharing of data chunks represented by different bins either. Hence, all the chunks belonging to a bin can be placed contiguously or in physical proximity of each other to speed up restore activities.

6.3 Unified Deduplication using Sparse Indexing with Min-hash Sampling

Traditional and non-traditional backup workloads, each can be deduplicated efficiently using technique designed specifically to exploit their salient features. Sparse Indexing

and Locality Sensitive Caching exploit locality to deduplicate traditional workloads. Extreme Binning uses file similarity to deduplicate non-traditional, file-based workloads. However, neither can deduplicate the other's workload without loss in performance.

Dedicated deduplication solutions, specifically designed for a given workload, require that separate systems be deployed to service each. For example, a Sparse Indexing box for nightly backups, an Extreme Binning solution for NAS-based clients, another solution for electronic mail servers. This is clearly not ideal. System administration could be much simpler if one system were able to handle all the workloads. Further, any duplicates present across different workloads could then be identified resulting in better storage space savings. Such a unified deduplication solution would *adapt* to workloads arriving from different sources, such as, VTL agents, electronic mail servers, NAS based shared file servers.

We tested Sparse Indexing and Extreme Binning for this purpose. Extreme Binning was not able to deduplicate large byte streams well. However, we found that Sparse Indexing, along with min-hash sampling, could be designed to adapt to a variety of workloads. We evaluated its performance on the basis of the deduplication quality and RAM usage.

Sparse Indexing chunks objects and samples the objects' chunk hashes to deduplicate them. These samples are called hooks. The sampling rate and object size dictate how many hooks are extracted per object. For larger objects, more hooks are extracted and vice versa. Though Sparse Indexing originally uses prefix-hash sampling, we found min-hash sampling to be more suitable. Min-hash sampling gaurantees the number of hooks expected given the sampling rate and object size. Further, we can enforce a minimum hook condition for objects that are too small to yield hooks.

We used real life data for experiments and generated a variety of workloads using it. We tested streamed archives, streamed files and individual file-based workloads. In addition, we also generated combination workloads consisting of a mixture of different workloads. We found that Sparse Indexing works well as a unified deduplication engine. Min-hash sampling is ideal for workloads which contain small objects as well as combination workloads because with min-hash sampling we can enforce a minimum hook condition which helps deduplicate even small objects.

Further, min-hash sampling yields slightly better deduplication than prefix-hash and though it uses slightly more RAM than prefix-hash, this overhead is close to 1% and, hence, not significant. When file boundaries are obeyed while chunking and metadata chunks are demarkated from content chunks, it gives us better deduplication. When one hook per object is enforced, it improves deduplication. However, the size of the sparse index almost doubles, meaning, RAM usage increases. If we use the enforced hook to only query the sparse index but do not add it to the sparse index, it is possible to improve deduplication without investing additional RAM.

These findings prove that using the above techniques, we can build an adaptable unified deduplication engine based on the Sparse Index design. Such a system is also easier to manage and administer than several dedicated systems deployed for each application/workload. Our experiments have shown that the unified deduplication solution also conserves more storage space than dedicated systems.

We have evaluated Sparse Indexing as a unified deduplication solution by measuring its RAM usage and deduplication quality only. Further evaluation of how its throughput would

be affected as it adapts to various workloads is required. The deduplication throughput for small objects is not expected to be as good as that obtained for large byte streams. The reason being that when deduplicating in bulk (10 MB objects at a time), the cost of every disk access gets amortized over larger data than it would when deduplicating, say, a 250 kB file. Further, placement strategies for small objects vs large ones must be studied. For example, should small objects and large objects be placed separately from one another even if they share content? If small objects arrive as a part of a weekly incremental backup, can their new chunks be placed with the chunks that arrived as a part of the most recent full backup for lesser fragmentation?

6.4 Introducing Strategic Redundancies to Improve Robustness

Chunk-based deduplication is a technique used widely in archival and backup systems systems. By removing duplicates or redundancies across objects it reduces storage space requirements. Some data, specifically backup data, contain a high number of duplicates. The storage space savings, sometimes up to 10 $\times$, clearly justify the resources invested into chunk-based deduplication.

However, there is an undesirable side-effect. Chunk-based deduplication introduces inter-file or inter-object dependencies. Duplicate chunks are shared across multiple objects. The loss of a shared chunk means the loss of all those objects since it will no longer be possible to reconstruct those objects in their entirety. The higher the degree of sharing, the greater the impact of such a loss. The loss of a chunk that is shared across many objects results in a large disproportionate loss of data. This means that deduplication, though desirable because of the storage space savings, can potentially reduce the robustness of the backup or archival system.

We have presented a simple strategy to increase the robustness of such data. We re-introduce strategic redundancies—we replicate certain important chunks so as to improve robustness. The degree of replication depends on the popularity of the chunk. Popularity is measured in two ways: the size of the data that depends on a chunk, and, the number of objects that share it. The size is the cumulative size of all the objects that share that chunk. Chunks that are highly popular are replicated more aggressively than chunks that aren't. For unique chunks, for example, only two copies are maintained.

Our goal was to control the balance between storage space savings and robustness. By re-introducing redundancies, we want to re-invest the storage space that we had saved via deduplication. We have used two real-world data sets for our experiments. One data set, the linux data set, represents the kind of workload you would expect in a backup system. The second data set was obtained from our local newspaper's archive. For both data sets our results show that our strategy gives a higher robustness *and* significant storage space savings compared with traditional LZ-based compression.

Our results show that our strategic replication approach can achieve a much higher robustness while using about half of the storage space required by mirrored LZ-compression. We can achieve close to 100% robustness for a small percentage of device failures. This higher robustness together with the savings in storage space is useful for future inclusion of repair models for backup and archival systems that use chunk-based deduplication. Our performance can only improve when we use other redundancy strategies such as RAID and erasure codes.

We conclude that by adjusting the number of replicas of individual data chunks based on our heuristics, backup systems and other long-term archives can reduce storage space re-

quirements while simultaneously increasing robustness, making the long-term storage of data both more affordable and more reliable.

6.5 Future Directions

To maximize the throughput of the deduplication process it is necessary to write new chunks to disk as fast as possible. New chunks are written contiguously while only references are updated for duplicate chunks. This causes fragmentation—all the chunks of an object are not placed contiguously on disk. Due to this, restoring an object may require a non-trivial number of random seeks. In the worst case, one disk seek per chunk, along with any additional seeks required for accessing the index/metadata to find the location of every chunk. This latency may be acceptable when restoring small objects/files or for archival data where the performance requirements are not very stringent. However, when the retrieval request is, for example, for a previous tape image which is typically hundreds of gigabytes in size, these disk seeks slow down the restore operation considerably. If this restore is required for crash recovery purposes such a latency is unacceptable. The degree of fragmentation depends on the fraction of duplicates in the data. Higher the number of duplicates, fewer the number of new chunks, and hence greater the fragmentation. This means that the most recent tape image will be more fragmented than the one backed up, say, a month ago. This also means that fragmentation gets worse with time. If most restore requests are for the more recently backed up tape images, then fragmentation will become a cause for concern. Most current products try to defragment data as a part of their housekeeping tasks, done when the system is not deduplicating data. However, the list of housekeeping tasks includes garbage collection, integrity checks, and in some cases, further

duplicate elimination as well. All these need to be done within a quiescent window of time, when the system is not deduplicating data. As the list of housekeeping tasks gets longer, it is crucial that every job be as efficient as possible. Defragmentation, however, is a very resource intensive operation. Hence, if it does not finish in time it may be throttled so as not to affect the performance of the primary deduplication jobs. Therefore, defragmentation as a post-process task scheduled after data deduplication will not be effective.

To solve this problem, we would like to co-locate an object's chunks at deduplication time, as opposed to a post-deduplication/housekeeping operation. To do this, some of its chunks, even if they are duplicate, will be re-written to prevent the object from becoming fragmented. For example, if more than n disk seeks are required to restore an object of size b bytes, then the object is deemed to be fragmented. One advantage of our approach is that since defragmentation is done during the deduplication itself, a separate defragmentation task will not need to be scheduled along with regular housekeeping tasks. A future direction is to to analyze this trade-off between storage space consumption and restore performance to measure the goodness of our approach.

Bibliography

[1] 104th Congress, United States of America. Public Law 104-191: Health Insurance Portability and Accountability Act (HIPAA), August 1996.

[2] 107th Congress, United States of America. Public Law 107-204: Sarbanes-Oxley Act of 2002, July 2002.

[3] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 36(SI):1–14, 2002.

[4] Miklos Ajtai, Randal Burns, Ronald Fagin, Darrell D. E. Long, and Larry Stockmeyer. Compactly encoding unstructured inputs with differential compression. *Journal of the Association for Computing Machinery*, 49(3):318–367, May 2002.

[5] Siddhartha Annapureddy, Michael J. Freedman, and David Mazières. Shark: Scaling file servers via cooperative caching. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, pages 129–142, 2005.

[6] Lior Aronovich, Ron Asher Ron, Eitan Bachmat, Haim Bitner, Michael Hirsch, and Shmuel T. Klein. The design of a similarity based deduplication system. In *SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, New York, NY, USA, 2009. ACM.

[7] Autodesk Maya. http://www.autodesk.com/maya. Maya is a registered trademark of Autodesk, Inc.

[8] Avamar Technologies Inc. http://www.avamar.com.

[9] Mary Baker, Mehul Shah, David S. H. Rosenthal, Mema Roussopoulos, Petros Maniatis, TJ Giuli, and Prashanth Bungale. A fresh look at the reliability of long-term digital storage. In *Proceedings of EuroSys 2006*, pages 221–234, April 2006.

[10] Deepavali Bhagwat, Kave Eshghi, Darrell D. E. Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proceedings of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '09)*, page To appear, September 2009.

[11] Deepavali Bhagwat, Kave Eshghi, and Pankaj Mehra. Content-based document routing and index partitioning for scalable similarity-based searches in a large corpus. In *KDD '07: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 105–112, 2007.

[12] Deepavali Bhagwat, Kristal Pollack, Darrell D. E. Long, Thomas S. J. Schwarz, Ethan L. Miller, and Jehan-François Pâris. Providing high reliability in a minimum redundancy

archival storage system. In *Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '06)*, pages 413–421, September 2006.

[13] Heidi Biggar. Experiencing Data De-Duplication: Improving Efficiency and Reducing Capacity Requirements. *The Enterprise Strategy Group*, February 2007.

[14] John Black. Compare-by-hash: a reasoned analysis. In *Proceedings of the Systems and Experience Track: 2006 USENIX Annual Technical Conference*, pages 85–90, 2006.

[15] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[16] Sergey Brin, James Davis, and Héctor García-Molina. Copy detection mechanisms for digital documents. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 398–409, 1995.

[17] Andrei Z. Broder. On the resemblance and containment of documents. In *SEQUENCES '97: Proceedings of the Compression and Complexity of Sequences 1997*, pages 21–29, 1997.

[18] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60(3):630–659, 2000.

[19] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29(8-13):1157–1166, 1997.

[20] Btrfs. http://btrfs.wiki.kernel.org.

[21] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[22] Joselíto J. Chua and Peter E. Tischer. Strategies for cooperative search in distributed databases. In *IAT '03: Proceedings of the IEEE/WIC International Conference on Intelligent Agent Technology*, page 325, 2003.

[23] Austin T. Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. Decentralized deduplication in SAN cluster file systems. In *Proceedings of the 2009 USENIX Annual Technical Conference*, January 2009.

[24] Brian F. Cooper. Guiding queries to information sources with infobeacons. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, pages 59–78, 2004.

[25] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 285–298, 2002.

[26] Peter B. Danzig, Jongsuk Ahn, John Noll, and Katia Obraczka. Distributed indexing: A scalable mechanism for distributed information retrieval. In *SIGIR '91: Proceedings of the 14th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 220–229, 1991.

[27] Data Domain. http://www.datadomain.com.

[28] Diligent Technologies. http://www.diligent.com.

[29] Fred Douglis and Arun Iyengar. Application-specific delta-encoding via resemblance detection. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 113–126, June 2003.

[30] Peter Druschel and Antony Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, page 75, 2001.

[31] Mike Dutch. Understanding data deduplication ratios. *SNIA Data Management Forum*, June 2008.

[32] *zlib* Compression Library. http://www.zlib.net.

[33] EMC Corporation. EMC Centera: Content Addressed Storage System, Data Sheet, April 2002.

[34] Kave Eshghi. A framework for analyzing and improving content-based chunking algorithms. Technical Report HPL-2005-30(R.1), Hewlett Packard Laboraties, Palo Alto, 2005.

[35] Kave Eshghi, Mark Lillibridge, Lawrence Wilcock, Guillaume Belrose, and Rycharde Hawkes. Jumbo Store: Providing efficient incremental upload and versioning for a utility rendering service. In *Proceedings of the Fifth USENIX Conference on File and Storage Technologies (FAST)*, pages 123–138, 2007.

[36] Ronald Fagin, Ravi Kumar, and D. Sivakumar. Comparing top k lists. In *SODA '03: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 28–36, 2003.

[37] George Forman, Kave Eshghi, and Stephane Chiocchetti. Finding similar files in large document repositories. In *KDD '05: Proceeding of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, pages 394–400, 2005.

[38] John F. Gantz, Christopher Chute, Alex Manfrediz, Stephen Minton, David Reinsel, Wolfgang Schlichting, and Anna Toncheva. The diverse and exploding digital universe: An updated forecast of worldwide information growth through 2011. An IDC White Paper - sponsored by EMC, IDC, March 2008.

[39] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of the 2004 International Conference on Dependable Systems and Networking (DSN 2004)*, June 2004.

[40] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005. USENIX.

[41] Val Henson. An analysis of compare-by-hash. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*, pages 13–18, 2003.

[42] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the Winter 1994 USENIX Technical Conference*, pages 235–246, Berkeley, CA, USA, 1994. USENIX Association.

[43] R. J. Honicky and Ethan L. Miller. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *Proceedings of the 18th International Parallel & Distributed Processing Symposium (IPDPS 2004)*, volume 1, pages 96–, 2004.

[44] Paul Jaccard. Étude comparative de la distribution orale dans une portion des Alpes et des Jura. *In Bulletin del la Société Vaudoise des Sciences Naturelles*, 37:547–579, 1901.

[45] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, September 1999.

[46] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 190–201, November 2000.

[47] Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 59–72, June 2004.

[48] Mark Lillibridge, Sameh Elnikety, Andrew Birrell, Mike Burrows, and Michael Isard. A cooperative internet backup scheme. In *Proceedings of the 2003 USENIX Annual Techni-*

cal Conference, pages 29–41, June 2003.

[49] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Campbell. Sparse Indexing: Large scale, inline deduplication using sampling and locality. In *Proceedings of the Eighth USENIX Conference on File and Storage Technologies (FAST)*, pages 111–123, February 2009.

[50] Witold Litwin, Marie-Anna Neimat, and Donovan A. Schneider. LH* — a scalable, distributed data structure. *ACM Transactions on Database Systems*, 21(4):480–525, 1996.

[51] Jie Lu and Jamie Callan. User modeling for full-text federated search in peer-to-peer networks. In *SIGIR '06: Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 332–339, 2006.

[52] Udi Manber. Finding similar files in a large file system. In *Proceedings of the Winter 1994 USENIX Technical Conference*, pages 1–10, January 1994.

[53] Petros Maniatis, Mema Roussopoulos, T. J. Giuli, David S. H. Rosenthal, and Mary Baker. The LOCKSS peer-to-peer digital preservation system. *ACM Transactions on Computer Systems*, 23(1):2–50, 2005.

[54] Gurmeet Singh Manku. Routing networks for distributed hash tables. In *Proceedings of the Twenty-First ACM Symposium on Principles of Distributed Computing (PODC 2003)*, pages 133–142, 2003.

[55] John McKnight, Tony Asaro, and Brian Babineau. Digital Archiving: End-User Survey and Market Forecast 2006–2010. *The Enterprise Strategy Group*, January 2006.

[56] Mario Mense and Christian Scheideler. SPREAD: An adaptive scheme for redundant and fair storage in dynamic heterogeneous storage systems. In *SODA '08: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete Algorithms*, pages 1135–1144, 2008.

[57] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 174–187, 2001.

[58] National Institute of Standards and Technology Secure hash standard. FIPS 180-2, August 2002.

[59] National Institute of Standards and Technology. Secure hash standard. FIPS 180-1, April 1995.

[60] Nexsan Technologies. http://www.nexsan.com.

[61] Oracle Corporation. Oracle Berkeley DB (http://www.oracle.com/technology/products/berkeley-db/index.html).

[62] Zan Ouyang, Nasir D. Memon, Torsten Suel, and Dimitre Trendafilov. Cluster-based delta compression of a collection of files. In *WISE '02: Proceedings of the 3rd International Conference on Web Information Systems Engineering*, pages 257–268, 2002.

[63] Claudia Pearce and Ethan Miller. The TELLTALE dynamic hypertext environment: Approaches to scalability. In Charles Nicholas and James Mayfield, editors, *Intelligent Hy-*

pertext, volume 1326 of *Lecture Notes in Computer Science*, pages 109–130. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0023962.

[64] Permabit Inc. http://www.permabit.com.

[65] Calicrates Policroniades and Ian Pratt. Alternatives for detecting redundancy in storage systems data. In *Proceedings of the General Track: 2004 USENIX Annual Technical Conference*, pages 73–86, 2004.

[66] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST)*, pages 89–101, 2002.

[67] Michael O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[68] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '01)*, pages 161–172, 2001.

[69] Sean Rhea, Russ Cox, and Alex Pesterev. Fast, inexpensive content-addressed storage in Foundation. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 143–156, 2008.

[70] R. Rivest. The MD5 message-digest algorithm. Request For Comments (RFC) 1321, IETF, April 1992.

[71] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[72] Gerard Salton. *Automatic Text Processing*. 1989.

[73] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. 1986.

[74] Hinrich Schütze and Craig Silverstein. Projections for efficient document clustering. In *SIGIR '97: Proceedings of the 20th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 74–81, 1997.

[75] Thomas J. E. Schwarz, Qin Xin, Ethan L. Miller, Darrell D. E. Long, Andy Hospodor, and Spencer Ng. Disk scrubbing in large archival storage systems. In *Proceedings of the 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '04)*, pages 409–418. IEEE, October 2004.

[76] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '01)*, pages 149–160, 2001.

[77] Storage Technology Corp. http://www.storagetek.com.

[78] The Internet Archive. http://www.archive.org.

[79] The Santa Cruz Sentinel. http://www.santacruzsentinel.com.

[80] Niraj Tolia, Michael Kozuch, Mahadev Satyanarayanan, Brad Karp, Thomas Bressoud, and Adrian Perrig. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 127–140, June 2003.

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

[82] Qin Xin, Ethan L. Miller, Thomas J.E. Schwarz, Darrell D. E. Long, Scott A. Brandt, and Witold Litwin. Reliability mechanisms for very large storage systems. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 146–156, April 2003.

[83] Lawrence L. You and Christos Karamanolis. Evaluation of efficient archival storage techniques. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, April 2004.

[84] Lawrence L. You, Kristal T. Pollack, and Darrell D. E. Long. Deep Store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, pages 804–815, April 2005.

[85] ZFS at OpenSolaris community. http://opensolaris.org/os/community/zfs/.

[86] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

[87] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST)*, pages 269–282, 2008.

[88] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.