Efficient Archival Data Storage

Technical Report UCSC-SSRC-06-04
June 2006

Lawrence You
you@soe.ucsc.edu

Storage Systems Research Center
Baskin School of Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064
http://www.ssrc.ucsc.edu/

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

EFFICIENT ARCHIVAL DATA STORAGE

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

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Lawrence L. You

June 2006

The Dissertation of Lawrence L. You
is approved:

Professor Darrell D. E. Long, Chair

Professor Scott A. Brandt

Professor Ethan L. Miller

Doctor Richard A. Golding

Lisa C. Sloan
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Efficient Archival Data Storage

by

Lawrence L. You

Archival storage systems must retain large volumes of data reliably over long periods of time at a low cost. Archival storage requirements and the type of stored data vary widely, from being highly compressed to highly redundant. The ever-increasing volume of archival data that need to be retained for long periods of time has motivated the design of low-cost, high-efficiency storage systems.

Due to economic factors, such as the rapidly decreasing cost of disk storage, memory, and processing—as well as improvements in technology, such as increased magnetic storage densities—research and development have moved toward disk-based archival storage. To further lower cost, they eliminate redundancy using inter-file and intra-file data compression. Each system uses its own strategy for compressing data, but none is consistently better than all strategies.

Our main contribution, presented in this dissertation, is to prove the thesis that it is possible to create a scalable archival storage system that efficiently stores diverse data by progressively applying large-scale data compression, providing better space efficiency than any single existing method. To support this, our work identifies common properties in these systems, evaluates efficient storage methods with respect to these properties, and presents a common framework in which data can be compressed using a combination of similarity detection and redundancy elimination methods. In addition, we have developed a prototype storage system using a *P*rogressive *R*edundancy *E*limination of *S*imilar and *I*dentical *D*ata *I*n *O*bjects (PRE-

SIDIO) framework. Similar and identical files are detected by the PRE algorithm. Data is recorded using a virtual content-addressable storage (VCAS) mechanism that can be used to store content with hybrid inter-file compression methods.

This work is a key part of the Deep Store archival storage architecture, a large-scale storage system that stores immutable data efficiently and reliably for long periods of time over a cluster of nodes that record data to disk.

Acknowledgments

I would like to thank the following people: Darrell Long, who has been my advisor for over a decade, for providing scholarly and practical advice. I am indebted to him for his patience and support, and have enjoyed sharing a learning experience that has been enlightening, engaging, and gratifying. My wife, Debbie Gravitz, whose support and encouragement are the reasons it has been possible to pursue this personal endeavor; and my sons, Davis and Nolan, who have been patient with their dad. My parents, Clare and Byong-Ki You, who have given me opportunity, encouragement, and freedom of pursuit throughout my life.

This dissertation wouldn't have been possible without the discussion, help, support, and friendship of all of the students, faculty, and staff at UCSC. I thank dissertation reading committee members Scott Brandt and Ethan Miller, as principal faculty of the Storage Systems Research Center who have joined with Darrell to grow the research group and evolve it into an enjoyable and productive environment; and to Richard Golding, who helped to pave the systems research path at UCSC before I arrived and to help me on the the way out. With Darrell and Scott, committee members Patrick Mantey and Alexandre Brandwajn, for approving my advancement to candidacy. Carol Mullane and Tracie Tucker, who were always available to help with all matters throughout my eleven years in the graduate program and willing to keep me in line whether my progress was steady or barely moving. My fellow UCSC graduate student, Tom Kroeger, whom I first met when he was an intern, has the distinction of giving me my first lesson at Santa Cruz—in surfing—the morning of my first class; I also thank him for his cameraderie and for letting me watch him navigate the PhD program before me; Randal Burns, for sharing his work and code on delta compression, and whose work we build upon; Ahmed Amer, who helped us secure an NSF grant and my advancement to candidacy; Kristal Pollack, for making the Deep Store project into a team instead of an isolated effort. Concurrent Systems

Laboratory, Computer Systems Research Group, Storage Systems Research Center, and Deep Store members including Andrew Wiser, Svetlana Kagan, Anuj Srivastava, Deepavali Bhagwat, and Kevin Greenan, for their discussions, ideas, implementation, and technical help. Ismail Ari, for fruitful discussions during our commute "over the hill" and sharing the experience of finishing the degree; Thomas Schwarz, for his guidance and discussions to understand problems in reliability. Larry Stockmeyer for his help in understanding aspects of data fingerprinting. The School of Engineering technical staff at UCSC for many years of support with the electronic infrastructure we all depend on.

Many friends and colleagues have given support in so many ways. I hope I can return you the favor as well. Special thanks to friends who have given their advice on PhD program as well as their wisdom and guidance of what was important and what to do next. My aunt, Youngbin Yim, who inspired me by starting her own PhD later in life to start a new career and then completing it much faster than I could have ever done. Tia Rich, who has provided guidance both for getting in and getting out of a graduate program; I wish everyone in a PhD program could have a friend like Tia who can bring clarity through introspection. Dave Burnard, Peter McInerney, and Neela Patel, who have given me perspective of their graduate experience in their respective fields.

My managers and colleagues at Taligent, Metrowerks, Pixo, and Google, for time and financial support they have provided directly or indirectly for my education at UCSC. Thanks to sponsors Hewlett-Packard Laboratories, CITRIS, Microsoft Research, UC MICRO, and the National Science Foundation for their financial support to myself and to the Deep Store project; and to Perforce, for providing educational licenses of their superb software configuration management system, which I have used to manage source code, presentations, papers, data, and of course, this dissertation.

With sorrow, I would like to dedicate this work to the late Larry Stockmeyer, Young-bin Yim, and Sidney Gravitz, my father-in-law, all whose work I have admired, who gave me encouragement during the period I worked on this dissertation, and with whom I would have liked to share the celebration of completing this degree.

Chapter 1

Introduction

We show that a large-scale archival storage system efficiently stores diverse data by progressively applying data compression algorithms in a single storage framework to provide better space efficiency than any existing storage compression method.

The amount of data that is created and that must be stored continues to grow [94]. Archival storage systems must retain large volumes of data reliably over long periods of time at a low cost. Archival storage requirements and the type of source material vary widely, from being highly compressed to highly redundant. For example, some digital audio and video is stored as compressed data and some not; in any case, file formats differ widely depending on use. Another example is achieving regulatory compliance, which involves storing many similar textual financial documents or email.

Archival storage systems store data efficiently by eliminating redundancy, but that comes at a cost. Data content must be analyzed, identical or similar data identified, and redundancy must be encoded or suppressed and also be retrieved reliably and quickly. Content analysis can use significant computing and bandwidth resources. Efficient storage systems produce internal storage metadata that is used to detect similar or identical data. However, if little

or no redundancy exists, the additional overhead of producing and searching the metadata is wasteful.

Prioritizing space efficiency in a storage system affects other areas of design. Reliability is improved by reintroducing redundancy, and it is also dependent on the degree of the interdependence of stored data. Query and content retrieval performance are directly related to the organization of stored data, which depends on data compression. Less tangible qualities are also important, such as the difficulty with which storage systems can be engineered to store data today as well as re-engineered to access data archives after generations of systems have passed.

In short, we want to store large volumes of immutable data permanently, efficiently, with high reliability and accessibility.

1.1 Contributions

Our main contributions are as follows. We measured storage compression efficiency and performance for different types of archival data and identified new types of data sets for computer storage systems. Using several archival storage methods, we identified common stages of data compression across a large corpus and measured their costs and benefits. We created a *Progressive Redundancy Elimination of Similar and Identical Data In Objects* storage framework, or PRESIDIO, which allows multiple content-addressable efficient storage methods to be integrated into a single system that balances space efficiency against read and write performance. Finally, we have designed the Deep Store archival storage architecture, which hosts an efficient content-addressable store such as PRESIDIO, to be scalable to a large scale system. We show that PRESIDIO stores data more efficiently that whole-file content-addressable storage, chunking, and as efficient as delta compression between similar files, using less resources than the most efficient single method.

In addition, we have made other contributions including: *chc*, a single file chunk archiving program; measuring degrees of inter-file dependencies for chunking and delta; measuring the relationship between dependencies and storage efficiency; and a preliminary observation of the trade-offs from introducing redundancy to improve reliability.

1.2 Evaluation

To evaluate the proof of our thesis, we examine several dimensions to the problem and evaluate them individually. Our examination starts with some existing solutions that address the problem of scalable archival storage, measures and analyzes experiments on synthetic and real-world data, presents the solution in the form of a design and architecture, and finally evaluates a prototype implementation. We briefly outline these individual evaluations below.

First, we evaluate algorithms that eliminate redundancy in storage systems. A number of existing methods for storing immutable data have been developed. However, no comparisons had been made in a manner that shows the absolute or relative storage efficiency, execution performance, or other measurements. We evaluate the existing methods to determine the common properties and steps, the performance we can expect, and the storage benefits.

Second, we evaluate the problem of addressing, compressing and recording data in a storage system in a scalable and unified manner. The question we try to answer is how we can store petabytes of data in a system using not just one compression scheme, but multiple. We assess a design and its ability to meet these goals.

Third, we examine and evaluate our solution to the efficient storage problem, the PRESIDIO storage framework and a prototype implementation. Storage space efficiency is our primary interest, but storage performance and bandwidth are also important for practical deployment of any solution. We measure compression rates, as well as computation and I/O bandwidth

to reveal the differences. Our analysis highlights trade-offs between the different methods, a key factor used in the PRESIDIO algorithm to improve compression while mitigating high resource usage.

1.3 Towards Efficient Disk-Based Archival Storage

Our research to develop an efficient archival storage system using hard disk was motivated by a number of economic, performance, technology, and social factors to meet increasing demands for storing increasing amounts of digital data. Our work starts with the premise that it is of practical benefit to use disk-based archival storage and then we set out to show how that can be done efficiently. We briefly describe the current computing and storage environment and then outline how our solution uses disk storage to help improve space efficiency of archival systems.

1.3.1 The Explosion of Digital Data

In recent years, the growth in the production of digital data and subsequent storage requirements have grown dramatically. In 2002 and estimated 92% of all new information was stored on magnetic media [94]. Despite decreasing tape storage costs [32], disk offers advantages that are increasing its adoption for archival storage. Disks allow new storage techniques that were not available, but they open up new challenges to research.

Reference data include content that is created and retained, such enterprise documents, scientific data, and communication forms in rich media formats such as web pages, email, messaging, photographs, and videos. Legislation such as the Sarbanes-Oxley Act of 2002 [165], whose purpose is to raise the level of corporate governance and improve corporate accountability, have had a tremendous impact on the storage industry as they meet the corpo-

rate need to retain documents for long periods of time. Email, financial records, and customer interaction account for significant amounts of data, mandated by section 404 for maintaining internal controls and procedures for financial accounting, and to store data in an immutable and tamper-resistant manner, as mandated in sections 802 and 1102.

Personal computing and consumer electronics technologies have also increased the demands for storage, for digitized music, digital photography, and digital video.

1.3.2 Demands in Computing on Archival Storage

Many industries and applications require access to more and more data with demands for shorter latency. Medical images, such as MRIs and X-rays, are being digitized with higher frequency. Media and news organizations would take advantage of an archival storage system by moving daily articles to on-line deep store. Enterprise organizations would benefit if corporate digital data, including user data, CRM (customer relations management), and web site or database assets were moved to deep store at low cost, and productivity would be improved by faster access with the additional benefit of operating high-availability, low-latency systems. Even individual computer users could benefit from a personal archival storage system to capture their own personal photographs, music, electronic mail, and records they create and use over time. In general, reducing the time to access data for applications that used to be stored on tape or optical disk reduces cost by reducing delays. And on-line availability of archival data that is easily distributed is critical in disaster recovery situations.

1.3.3 Advances in Storage Technologies

The storage industry has introduced new technologies to meet growing demands from increasing data volumes. High volume solid state flash memories, recordable CDs and DVDs

and digital video tape are capable of storing high quality digital data. But for archival storage, these improvements have not been as consistent or dramatic as magnetic hard disk technologies. Disks have shown promise of increasing apparent data capacity through data compression beyond the capabilities of streaming storage.

Data compression performance has improved in the last few years by using differential compression techniques that create delta encodings of data related to other stored data [4, 80]. These algorithms execute efficiently and compress at rates over 100:1 (or 99% reduction in space) for data sets with high redundancy and encode differences with *add* and *copy* operations. The typical usage of these techniques is to compress versions of files, but this typically requires at least one file to be available on-line, with lower latency than is possible with traditionally slow tertiary storage. Higher rates of compression yield higher effective bandwidth of storage devices; this allows inexpensive disks, when combined with differential compression, to be even more economical than tertiary storage.

The key improvement hard disks offer is the opportunity for inter-file compression as well as intra-file (or streaming) compression. Existing archival storage technologies, magnetic tape and optical disk, do little to exploit the new technologies in providing high degrees of compression, or to aid in information retrieval. The shortcoming of linear storage media such as tape are that they provide one-dimensional access. Optical media exhibits better random access behavior than tape, but not nearly as good as hard disks.

Worse, tape and optical costs have not dropped in cost as dramatically as magnetic disk in the last three years. In 1999, tape media to disk cost ratios per byte were approximately 1:16 [57], but today the ratio is closer to 1:1.5, and that excludes the cost of a tape drive.

1.3.4 Moving Away From Tape

Economic and technological factors support the trend of the transition from traditional archival media to magnetic disk based-storage. In dramatic fashion, disk costs have decreased at the same time storage density have increased. In 1999, Gray predicted that "within five years . . . the prices should be nearing 1 K$/TB" [57]. In fact, we have seen the price of storage devices below that threshold.

Backup technology is no longer good enough for archival storage. Tape has comparatively high latency over hard disk; near-line data retrieval using tape robots takes approximately 2–4 minutes [105] and the utilization of tape drives has been measured at 15% [107]; the rest of the time is spent positioning the tape. Retrieving data from off-line archival storage requires human intervention and even longer delays. With traditional deep store systems using tapes, differential compression as it is used in our architecture is impractical, since it requires random access to file data, but with our disk-based approach it is possible to achieve new levels of compression. Because of the additional overhead to access tapes, the integrity of data stored on tape and optical disks is not easily tested.

1.4 Organization

Our study in storage system design explores and develops novel techniques into a foundation for large-scale storage systems. The main theme of this work is data compression over large volumes of data. Its purpose is to map out the similarities, to identify the differences, and then to exploit those properties in an integrated storage strategy. Figure 1.1 illustrates the major topics of this study and their relationships, which are *feature selection*, *similarity detection*, *redundancy elimination*, and *content-addressable storage*, assembled into a space-

efficient storage framework. The fifth topic, *efficient storage methods*, lists the algorithms with different properties that are applied progressively in PRESIDIO to reduce storage usage.

The figure is a road map for this dissertation and a representation of the integrated solution to our thesis. It shows the dependency graph of relationships consisting of abstract concepts and concrete algorithms or methods, and concrete instances of implementations and references to instances. Concrete algorithms point to abstract concepts by arrow. For example, the *digest* is a concrete algorithm based on the abstract concept of *bit string hashing*. Similarly, references to instances of data or algorithms are connected by a line terminated by a dot. We build the architecture from a number of concepts, ranging from hashing, data fingerprinting, chunking, and shingling within the topic of feature selection. The topic of similarity detection includes digest hash retrieval and chunk digest hash retrieval that identify exact data, and similar file detection using fingerprints and superfingerprints that use approximate resemblance to detect similar data. Redundancy elimination includes various data compression algorithms including inter-file, intra-file, and hybrid compression strategies. The content-addressable storage topic covers the low-level implementation for storing actual bits onto devices. Finally, the efficient storage methods tie all of the topics together into a single framework that uses hybrid methods.

In Chapter 2, we provide the background to the problem and related work. Chapter 3 is an overview of our solution, PRESIDIO, which includes a description of a prototype implementation and a summary of our evaluation. It describes relationships between the main phases, illustrated as the boxes in Figure 1.1. Chapter 4 describes and evaluates content analysis algorithms and data structures in *feature selection*. Chapter 5 describes and evaluates methods for measuring similarity and finding similar data. Chapter 6 describes redundancy elimination methods. Chapter 7 describes the unified content-addressable storage subsystem used by PRE-

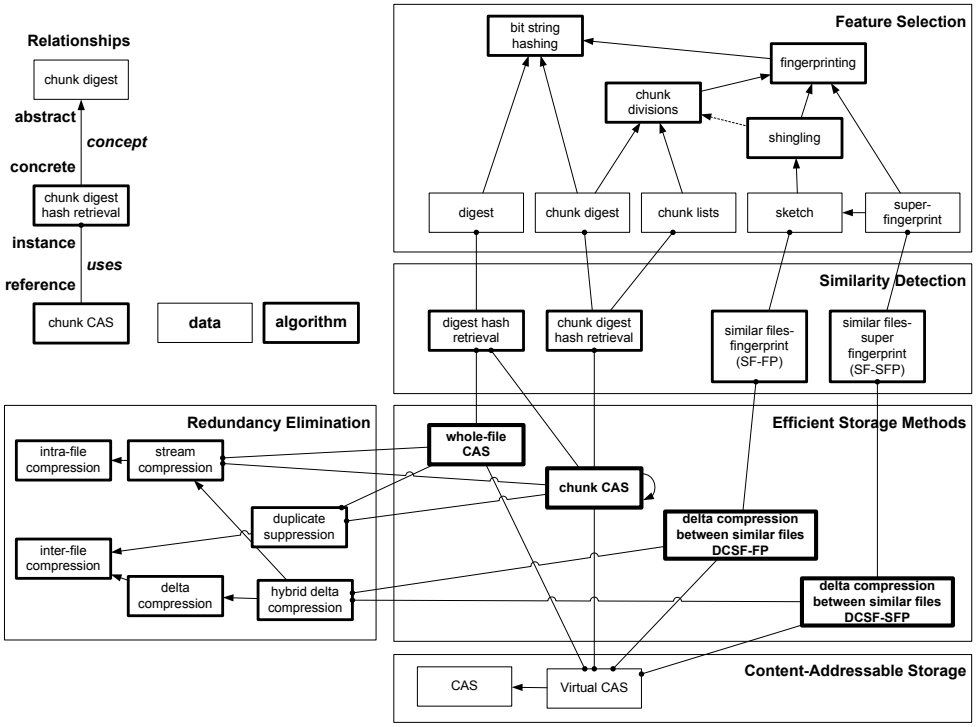


Figure 1.1: Overview of the algorithmic and data relationships in a space-efficient storage system

SIDIO to record and reconstruct data. Chapter 8 describes the algorithms to progressively compress data using the PRESIDIO algorithm and framework. In conclusion, Chapter 9 summarizes our work and briefly discusses avenues for future work. Appendix A contains implementation and technical details of the Rabin fingerprinting by random polynomial hashing method, a hashing function used throughout our work.

Chapter 2

Background

The field of archival storage research covers decades of changes in technology and orders of magnitude improvements in technology and cost. Like many problems in systems research, we draw from a long history in many areas to formulate a new solution that is possible from recent changes such as the lowered cost of disk storage, the increased density of memories, and high availability of cheap computation capabilities.

We describe the background in this area of storage research with the following organization. First, we describe the problems facing archival storage today. Next, we describe our solution to the problem to efficient storage as it relates to a larger project, Deep Store, in which we are attempting to build scalable, reliably, and accessible storage in a larger setting. Finally, we cover the many fields of research and related work.

2.1 Fulfilling the Need to Archive Data

Today, the need for large-scale storage systems is obvious; a study estimated that over five exabytes (5×2^{60} bytes, or about 5×10^{18} bytes) of data was produced [94] in 2002,

an increase more than 30% over the previous year. Furthermore, the fraction of data that is *fixed content* or *reference data* continues to increase, accounting for 37% of all stored data, and was expected to surpass mutable data by the end of 2004. This is unsurprising, especially in the face of the over 10,000 legal regulations placed on companies in the U.S. for corporate compliance [157]. The storage for compliance increased by 63% just in 2003, even before some of the most demanding regulations, such as the Sarbanes-Oxley Act, went into effect.

Scientific computing simulations, life sciences databases, and environmental data represent huge volumes of data. Several databases range in volume from terabytes to petabytes [61].

Further compounding the need for archival storage is the increasing volume of material converted into the digital domain. Permanent records archives, in which data is not removed, will only continue to grow. The National Archives and Records Administration (NARA) aim to have 36 petabytes of archival data on-line by the year 2010. As a result of these increased demands, reference storage, for both archival and compliance purposes, is a rapidly growing area of interest [156].

Despite the plummeting cost of low-cost consumer storage devices, the cost of managed disk-based storage is high—many times the cost of a storage device itself and higher than tape. In 2002, the cost for enterprise disk storage was over $100 per gigabyte, compared to tape at $10 per gigabyte. A trend for near-line and archival storage is to use cheaper disks, such as ATA devices, instead of SCSI devices, in order to bring down storage cost closer to that of magnetic tape [57, 56].

In 1989, Katz *et al.* cite the predicted maximal areal density (MAD) by "the first law in disk density" [77]:

$$MAD = 10^{(year-1971)/10} \quad \text{megabits per square inch} \quad (Mb/in^2) \tag{2.1}$$

However, since then, disk density has far surpassed those estimates, and in recent years has increased at a higher rate. In 1999 IBM Deskstar 37GP (3.5 inch, 34 GB, 7200 RPM) drive [71] areal density was 5.15 gigabits per square inch (Gb/in^2), but the projection using the formula above should have been $10^{(1999-1971)/10} = 10^{28/10} = 631$ Mb/in^2, an error of more than 8 times. More recently, in 2004 Hitachi introduced the Ultrastar 10K300 300 GB (enterprise class SCSI) and Deskstar 7K400 (consumer grade ATA/SATA 3.5 inch, 400 GB, 7200 RPM) disk drives [62] that are available today are even more dense, at $61-62$ Gb/in^2, when the projection shows 2 Gb/in^2, or an error of more than 30 times.

A new class of storage systems whose purpose is to retain large volumes of immutable data is now evolving. The engineering challenges include: improving scalability, to accommodate growing amounts of archival content; improving space efficiency, to reduce costs; increasing reliability, to preserve data on storage devices with short operational lifetimes and inadequate data integrity for archival storage; and locating and retrieving data from within an archival store. High-performance disk-based storage designs have been evolving to use lower-cost components, but they continue to be expensive to manage.

The trade-off between space efficiency and redundancy for reliability has always been an issue; however, for archival systems the trade-off is deepened. By the very nature of archives, they will continuously grow over time because data is rarely removed. This creates the need for a space-efficient solution to archival storage. However, the need for a reliable system is also heightened in the case of archival storage since as stored data gets older it is more likely that there will be undetected cases of bit rot, and as devices age the likelihood of their failure grows. Efficient archival storage systems pose a problem: existing reliability models do not consider metrics or have the means to compute failure of recorded data based on the degree of data dependence.

2.2 Deep Store: Scalable Archival Storage

To address these problems, we have proposed a storage architecture for the Deep Store archival storage system, designed to retain large volumes of data efficiently and reliably.

Traditional disk-based file systems, which include direct- or networked-attached storage (DAS/NAS) and storage area networks (SAN), do not have the properties desirable for archival storage. They are designed to have high performance instead of a high level of permanence, to allocate data in blocks instead of maximizing space efficiency, to read and write data instead of storing it immutably, and to provide some security but not to be tamper-resistant.

Archival data must be retained for retrieval after a period of time that exceeds the life expectancy of disk-based storage systems hardware and likely to exceed the practical lifetime of the storage system software and their interfaces. Digital data must be stored reliably and automatically managed by the storage system in order to be preserved beyond single failures. In some cases, data lifecycles may require practices to destroy certain content after a period of time. For instance, corporate cost and legal liability is reduced if email or non-final documents are destroyed according to pre-determined policy [122].

We desire the following properties in an archival storage system that set it apart from file systems: significantly reduced storage cost, immutable properties (write once, read many), cold storage (write once, read rarely), dynamically scalable storage (incremental growth of storage), improved reliability (checksums, active detection, preferential replication), and archival storage compliance (WORM, required duration, lifecycle management).

Additional properties to distinguish Deep Store from other archival storage systems include: much lower latency than the tape systems which it replaces, a simple interface and design, searching capabilities (essential to petabyte-scale storage systems), and accessibility across decades or centuries as well as across local or distributed systems.

At the foundation of Deep Store is PRESIDIO, the efficient storage subsystem. Its architecture presents a simple content-addressable storage interface, a compression algorithm that progressively applies and evaluates multiple methods to yield storage efficiency, and a low-level virtual content-addressable storage framework that encodes and decodes content depending on the storage method used.

2.3 Space-Efficient Storage Systems

Archival storage systems that reduce redundancy have a long history of storing data cheaply for long periods of time. These systems typically have been at the end of the memory storage hierarchy: *primary storage* in the form of random-access main memory, *secondary storage* in the form of random-access magnetic disk, and finally *tertiary storage* in the form of magnetic tape or optical disk. Hierarchical storage management spans these systems by automatically migrating files through the hierarchy [53].

Traditional tertiary storage media, especially tape, do little to eliminate all redundancy due their physical constraints: they are good at sequential access but poor at random access. This left storage designers to extract the most out of stream-based compressors. IBM ADSTAR Distributed Storage Manager backup system (ADSM; and known as IBM Tivoli Storage Manager) [70] pioneered the use of delta compression in tape-based backup; its shortcoming is that redundancy is eliminated for versions of files, and not across the entire data set being backed up [20]. Furthermore, the high latency for accessing tapes forced delta chains—the length of the dependency graph formed from delta compression operations—to a small length; *version jumping* [20] helped limit chain length at a modest cost of efficiency.

However, inter-object and inter-file redundancy can be significant and a more current area of interest due to the potential for reducing storage cost. We briefly cover recent work that

eliminates redundancy in storage.

2.3.1 Chunking: Dividing Data Streams by Content

Streams of data can be divided into non-overlapping strings of contiguous data, called *chunks*. Subdivision is useful for identifying instances of identical data. Spring and Wetherall developed a technique to eliminate redundant network traffic [151] by dividing data into chunks in a manner similar to the one used by Manber [100]. Our work evaluates the chunking technique first used by the Low-Bandwidth Network File System (LBFS) [110]. To eliminate unnecessary network transmission, both client and server avoid sending data chunks if they are already found in a cache on the other end. Data chunks are delimited by *breakpoints* located deterministically from the data content. One such method is to compute a hash function over a sliding window of a few tens of bytes and then to select the breakpoint when the hash value meets a simple criterion, for example when its integer representation modulo an integer constant is zero. LBFS computed the hash over the window efficiently using Rabin fingerprints of 48-byte length and selecting special values of those fingerprints. Next, data chunks were identified by their SHA-1 hash values between delimiting breakpoints.

The *rsync* program synchronizes remotely located files efficiently [164]. The algorithm identifies identical substrings of data in two locations in the following manner. First, a *fast signature* algorithm computes a fingerprint over a sliding window. Second, a *strong signature* computes a hash over a block, such as the MD4 digest, a 128 bit hash value [132]. To efficiently process a file, the fast signature incrementally computes over every position in the file with a small number of instructions, and the strong signature computes a digest over each block of size in the range of hundreds to thousands of bytes. The small amount of computation in *rsync* yields high throughput and significantly reduces transmission of data. However, the al-

gorithm compares specific pairs of files to compute and then transmit differences. In an archival storage system, there are two shortcomings of the *rsync* method for redundancy elimination in an archive. First, a client would first be required to identify a similar file that had been stored previously in order to minimize the transmission of differences between two files. This precondition would be satisfied in a versioning file system, but not in a general solution. Second, differences in *rsync* are computed with a fixed block granularity, unlike a delta compression program that would identify common substrings within a file of arbitrary length.

2.3.2 Delta Compression and Resemblance Detection

Our work seeks to improve storage system efficiency by eliminating identical or similar data across files. Since whole files or large contiguous regions of files may only be similar and not identical, we turn to delta compression to express file encodings as differences between two files. Douglis and Iyengar evaluated the efficiency of storage by using delta encoding via resemblance detection (DERD) over a number of parameters, showing the relationship between similarity and delta encoding sizes, and establishing a number of parameters such as resemblance thresholds, feature set sizes, and the number of delta encodings to compute to determine the best match.

Manber used fingerprints to determine file similarity on a single volume, computing the entire contents of the volume as a single operation or incrementally as new files are changed [100].

Fingerprints were computed between special or random *anchor* values, and identical subsections of files were found between those anchors when the fingerprints matched. The time and space to compute the similarity of files varied with the number of fingerprints per file as well as with the number of files. The design of this system was not scalable without using

unlimited resources and required sorting and managing large sets of data.

Likewise, research in discovering web similarity has been of keen interest in recent past. More recent studies have shown that there is a significant amount of duplication of web documents [15, 39, 41]. For example, an April 1996 web walk of over 30 million web pages generated clustering data: using a cluster resemblance of 50%, 3.6 million clusters were found containing 12.3 million pages. However, 2.1 million clusters contained only identical pages (5.3 million) and the remaining 1.5 million clusters contained 7 million pages which were either identical or similar. A later experiment showed that 22% of web documents were identical and 48% were similar [148]. Using a separate method, later experiments showed the web grew to 24 million pages; identical pages numbered approximately 22% and with some similarity, to a total of 48% [148].

Web searches over millions or billions of documents pose challenges that overlap with the problem of detecting similar or identical data in file systems. Some difficult problems in web search include the detection of related pages [59] and finding similar data in high-dimensional search spaces [72]. The properties that make these techniques less suitable for file system storage systems are updates and indexings of the corpus that are less frequent than the rate of ingest in an archival store, the lack of a requirement for index space efficiency, and the nature of structured natural language hypertext documents, which are more suited for extracting features that are in the same namespace as the textual search terms.

Other techniques to detect duplication of documents have been useful but not deployed on an extremely large scale [147, 47, 64]. Most research to detect similarity uses fingerprinting techniques as well.

The redundancy elimination at the block level (REBL) scheme improves on the performance of DERD by using superfingerprints to detect similar data [83]. A number of data sets

were measured using different compression methods to highlight the effectiveness of superfingerprints, *i.e.* fingerprints of features selected from files. When superfingerprints are used to detect similar blocks in highly similar files, REBL compresses well, but for larger data sets of less similar files, the benefits are not as evident. Measurements focus on the important step of resemblance detection, however computing and selecting a large set of fingerprints for the purpose of fingerprinting must also be accounted. Overhead for file metadata and efficient storage metadata are not measured and incur an additional, but necessary cost to storage.

Cluster-Based Delta Compression of a Collection of Files [118] evaluates delta compression across a corpus of data. Their work computes differences between data to compute efficient delta encodings, but does not address the storage of data incrementally.

2.3.3 Content-Addressable Storage

Several storage systems use *content-addressable storage* (CAS) to identify files by content. *Content-Derived Names* (CDN) was early example of a CAS-like system using probabilistically unique hashes to identify files for configuration management [65].

Current archival storage systems are commercially available and under research, typically configured as network attached systems that are used in a manner similar to more traditional network file systems. EMC Corporation's Centera on-line archival storage system [42] uses CAS, identifying files with 128-bit hash values. Each file with identical hash is stored just once (plus a mirror copy for fault tolerance).

Venti [125], provides archival storage with write-once characteristics. Venti views files as fixed-size blocks and hierarchies of blocks that can help reconstruct entire files. Identical blocks are stored only once and differences are computed at the block level. Fingerprints are generated from blocks and from hierarchies of blocks. Versions of files are stored at fixed

locations and common data is not shared unless they are identical on fixed block boundaries.

Storage Tank, an IBM SAN file system, employed duplicate data elimination (DDE) over mutable data by coalescing blocks that were identified using content-addressable storage [66]. Like Venti, only duplicate blocks were identified by hash (SHA-1). The finer granularity of block hashing achieves a higher level of block reuse over whole-file hashing and avoids extra hashing over unchanged blocks. The use of fixed-block sizes avoids excessive recomputation incurred to detect variable-sized block boundaries. Copy-on-write allows in-place modification with low write cost, and duplicates are detected using a lazy update.

Content-addressable storage can be extended to data of arbitrary length. Chunks, whose boundaries are defined by content, divide files into smaller pieces. Chunks are also identified and stored by their content address. Commercial products from Avamar and Rocksoft use variations of this method, as well as other research projects [45].

2.3.4 Versioning File Systems

Versioning file systems such as the Elephant file system [140] and the Network Appliance Write-Anywhere File Layout (WAFL) [63] (also employed in their NearStore product [112]) are similar to archival storage systems in that they retain multiple versions of files. However, they do not have the ability to retain all copies permanently or to eliminate redundancy across files. Because versioning file systems typically operate on snapshots of read-only data, efficient archival storage systems enable new opportunities for their use if incremental storage requirements are much smaller than sharing whole files. In addition, versioned file systems do not exploit similarity of files and at most share blocks [125].

2.4 Scalable Storage Systems

Scalable storage systems use distributed resources to improve total storage capacity, to distribute read/write bandwidth over multiple network interfaces and storage media heads, and to decentralize control to reduce bottlenecks. This category includes *cluster file systems*, *distributed systems*, and *peer-to-peer systems*.

The IBM General Parallel File System (GPFS) [142], a cluster file system, distributes storage over multiple nodes. A single directory and file namespace is shared across computers connected via high-speed LAN. Files are stored with larger block sizes, from 16 KB to 256 KB, and the blocks are distributed across nodes. Replication and failure are handled by distributing data on different nodes so that a single failure would not cause its owning cluster to fail. In a similar manner, the Google File System (GFS) [51], incorporates replicated redundant servers to provide reliability. Large blocks, called *chunks*, are 64 MB in size. GFS is designed to work well with the *append* operation. Scalability is achieved with large numbers of slave machines, or *chunkservers*. Storage reliability is greatly improved through replication and migration. Availability is improved through failover from master servers to replicas. Bandwidth is increased by reading and writing directly to different chunkservers that store copies of a chunk. In large cluster file systems, CPU and memory are available on each node and help manage metadata and assist distribution. Larger block sizes are necessary to reduce metadata overhead. The properties found in these on-line high-performance mutable storage systems helped influence the design of our proposed system architecture. Large block sizes and limited metadata storage make GPFS and GFS suitable for smaller numbers of large files, however archival data does not always meet this criteria.

Distributed storage systems include cluster file systems, storage area networks (SAN). These include virtual disk storage systems like Petal [84] and Frangiapani [158]. Distributed

backup systems like Pastiche [37] provide efficient distributed storage when common data exists across multiple machines. These systems do little to improve space efficiency but may provide a model for improving data distribution in an efficient archival storage system.

Peer-to-peer storage systems further decentralize storage, even to the point where control is decentralized. Such self-organizing systems may be tolerant of failures, network segmentation, attacks to compromise data. When these peer-to-peer systems are widely distributed, they can also exploit bandwidth in a wide area network to provide high aggregate bandwidth. Naming and locating data is usually dependent on an *overlay network* such as Chord [153], Pastry [136] or CAN [129]. Examples of peer-to-peer storage include PAST [137], Pasta [108], Silverback [168], Herodotus [17], and Archival Intermemory [54, 35].

2.5 Lossless Data Compression

Our work incorporates existing data compression methods into a framework that uses complementary methods and balances space efficiency with compression performance. We describe commonly available methods and their place in our research.

We are primarily interested in *lossless data compression*, which preserves input data exactly. File contents are presented for storage and at a later time, the exact contents are retrieved. The *lossy data compression*, which is commonly used to encode audio or visual content such as pictures, sound, and movies efficiently, is a related problem which we discuss briefly below.

2.5.1 Stream Compression

Lossless *stream compression* takes an input stream and emits a (generally) smaller output stream by eliminating redundancy that is seen within the stream itself. Input symbols

are converted to codes and then output. Codes, which are of varying size, are selected statically or dynamically and assigned to represent symbols depending on the probability of their occurrence [111, 141].

Stream compression exhibits distinct properties. The streaming data window over which the compressor can evaluate the statistical properties is typically limited to real memory and often is much smaller. Some compression programs maintains a dictionary that is updated as symbols and strings are scanned from input. This prevents stream compression from being effective across files and even within large files which may have multiple instances of common data separated by large distances.

Stream compressors can encode identical data with different codes. Static coders, such as Huffman encoding, depend on the input. Dynamic coders, which update encodings as the probability of symbol and substring appearance, adapt to input. In either case, unless a static coder is also using a permanently assigned encoding, the encoded (and compressed) data is not guaranteed to contain identical regions even if two uncompressed regions are identical. Therefore, stream compression can interfere with inter-file compression if it is applied early. The XRAY system used a variation of a pure stream compressor whereby an additional training pass would create a semistatic model of phrase-based probabilities, separating the phrase dictionary from individual files [30]. The inter-file training helps compress collections of similar data but is fundamentally a stream compressor.

Performance of stream compressors is very good and can be implemented in hardware. The current generation of processors, *e.g.* Pentium 4 at 2.5 GHz, also offer good software implementations; we measured compression at rates in excess of 200 megabytes per second, faster than the approximately 50 megabytes per second streaming read/write performance of a single hard disk device. Many data compression implementations exist including the pop-

ular programs *zip*, *compress*, *lharc*, *gzip* [49] (based on Lempel-Ziv compression [177]), and *bzip2* [145] (based on Burrows-Wheeler compression [24]).

2.5.2 Delta Compression (Differential Compression)

Delta compression, or *differential compression*, is a method for computing differences between two sources of data in order to produce a small *delta* encoding that represents the changes from one data source to another. Generally, these data streams contain arbitrary content (*i.e.* not just text) and are files. Delta encoding has been used for storage, for in-place updates of data [22], and for reducing network bandwidth usage [104].

Delta compression programs scan two sets of input files, the first set consisting of one or more *reference files* or *base files*, and the second a single *version file*. The program scans the input files, computes a difference to the version file and then produces a *delta file*. The reference and version files are usually related, most often as instances of changes from one version to the next, and stored in the same file system location but differing in time. Otherwise, reference and version files are selected for input explicitly based on an *a priori* relationship. Delta compression is one of the two classes of inter-file compression algorithms used in our solution.

The methods to compute differences between files are tailored to file contents. Early programs to compute differences operated on text files. One commonly used program for computing differences still used today is the *diff* [69] tool, used in the RCS version control system [160]. Differences were computed at the line level, unlike binary delta, which does not have well-defined characters defining units of text. Later programs computed binary differences. Reichenberger described an algorithm for differencing binary data [130]. This *greedy* algorithm can produce an optimal (smallest) delta encoding, but exhibits quadratic execution

time in the size of the file inputs and uses memory space linearly in proportion to the size of the input files.

The *edit distance* [159], or *Levenshtein distance*, is one measurement of the difference between two files or documents as a function of the number of insertions, deletions or exchanges of characters. Our work uses delta compression, which aims to reduce edit distance. Information theoretical differences such as the Chaitin-Kolmogorov difference might result in smaller algorithmic representations of operations to transform one file into another; however, computing optimal differences requires high computational complexity and are more useful in theory than in practice [44]. For heuristic metrics, the weakly approximate edit distance [6] may be a practical implementation for use of measuring resemblance, but not necessarily for easily detecting resemblance of a single file against a large collection of files.

Burns improved delta compression performance by applying heuristic techniques to compute the delta encoding in linear time, using constant space [21, 19]. The algorithms are known as *One Pass* and *One and a Half Pass*. A more complete analysis of the execution performance of these algorithms can be found by Ajtai, Burns, Fagin, Long, and Stockmeyer, [4].

When the type of binary data is known, specialized differencing can be applied. Examples of programs that compute deltas between versions of executable code include *BSDiff* [121], *RTPatch* [123], and *Exediff* [5].

2.5.2.1 Reconstruction

File reconstruction from delta files is straightforward by applying *copy* and *add* operations from a *version* (or *delta*) file to a *reference* (or *base*) file. When a *reference* file needs to be first reconstructed, the complete reconstruction depends on all antecedent files. This recon-

struction may incur a number of resource costs, including computation, memory, or intermediate storage.

Yu and Rosenkrantz reduced the retreival time of the classical scheme for reconstruction versions, whose worst-case execution time was quadratic in the sum of the size of the initial complete version and the sizes of the file differences down to a reconstruction time that is linear in the sum of the size of the initial complete version and the sizes of the file differences [175]. While the cost of the reconstruction can be bounded from the file sizes, this scheme assumes that the files are immediately available.

When the time to retrieve *reference* and *version* files is non-trivial, other methods for storing these files can take less time. For instance, when files are stored on *tertiary storage* such as magnetic tape (whose retrieval latency can take many seconds, minutes or even days), a method described by Burns and Long known as *version jumping* [20] can be used to reduce the number of file retrievals, thus reducing the load at the server and also reducing network transmission time.

Several delta programs are available today, including *diff*, *bdiff*, *vdiff*, *vcdiff*, *xdelta*, *zdelta*, *bsdiff*, *dfc-gorilla* and *RTPatch*. In a 1998 study, Hunt, Vo, and Tichy compared delta algorithms by measuring performance over 1,300 pairs of files from two successive releases of GNU software [68] using the *diff*, *bdiff*, and *vdiff* programs. Since that time, other implementations have appeared, such as *vcdiff* [80] (an IETF standard [79] and successor to *vdiff*), *xdelta* [95, 96], and *zdelta* [162].

A comparison of delta compression programs is reported in the *zdelta* [162] technical report, measuring storage efficiency, compression and decompression speed of *gzip* (a file compression program) against *xdelta*, *zdelta*, and *vcdiff*. The report outlines an experimental methodology for producing synthetic test files varying in similarity from 0 (completely dis-

similar files) to 1 (completely identical files). The main difference in performance is in the compression encoding schemes that are used in each program: *xdelta* computes a difference but does not encode it, *vcdiff* uses a byte-based encoding, and *zdelta* encodes a difference using Huffman codes. The zdelta program uses *zlib* [50] lossless data compression library based on the LZ77 algorithm [177].

2.6 Lossy Data Compression

Although our work focuses on lossless data compression, many data types and media forms are stored using *lossy data compression*. Sampled analog input data, such as audio, still images, and motion pictures are encoded using a wide range of data compression algorithms such those defined by JPEG (Joint Photographic Experts Group), JBIG (Joint Bi-level Image experts Group), MPEG (Motion Pictures Expert Group), and DV (Digital Video), to name a few. The compression algorithms produce encoded data that when uncompressed, are near replica digital data of the original input, but exhibit small errors.

When lossless image compression is applied by different users, different output can result despite small perceptive differences to the users who see or hear these data. For example, searches conducted on search engines for images may produce large numbers of identical or nearly identical images based purely on their metadata. Detecting similarity in such compressed data is not likely to be further compressed, but some compression may be possible if similarities are detected on the uncompressed data. Because the uncompressed data for media content is interpreted by human perception and the compression methods seek to minimize errors in perception, lossy data compression in storage systems is an topic for investigation outside of our research area.

2.7 Document Similarity

In the field of study for *information retrieval* (IR), *document similarity* is a metric that is often used to measure the similarity of two or more textual documents. Similarity is commonly used by query operations, ranking the results and assigning a similarity measure to each document that is retrieved [170].

One commonly used method to compute document similarity is the *cosine measure* within the *vector space model* [8] . Documents are represented within a vector space with dimensions d, the number of documents, described by t terms, a $t \times d$ matrix A. Each element a_{ij} represents a weighted frequency in which the term i occurs in the document j. Rows, representing documents, are also called *document vectors*. Queries are represented as *query vectors*, using the same terms found in the documents. Treating the vector space as Euclidean, the similarity is computed as the cosine of angles the query and document vectors. Query results are considered when the cosine is larger than a specified threshold. For natural language text documents, terms are words. Relative word frequency within a document improves the weighting of a term; therefore, documents with higher occurrences of a word will have a higher likelihood of matching a query.

Word frequency may be useful for finding related documents, but it is not clear that it has any relation to redundancy that can be extracted between the two documents. For instance, two documents each with a random ordering of words with the same term frequency will have a very high cosine, but there will be little inter-document redundancy to eliminate and exploiting the apparent similarity in an efficient storage system is not likely.

Arbitrary binary files exhibit a similar problem: the terms, unlike documents composed of words, are not defined by one convention such as natural language. The number of terms in a natural language are limited to a subset (through *stemming*) of all words used in

the language's vocabulary; conversely, binary data would have an arbitrarily large set of terms. The dimensions of the already sparse vector space grow unbounded and therefore the burden of managing and searching through such space creates a new problem for storage efficiency.

File types, often specified by filename extension or file preamble, define contexts in which terms can be defined. Our research assumes those distinctions are not necessary; however, using a secondary vector space model to further detect similarity may prove useful in future work. Vector space models have been used to search peer-to-peer networks [155] as well as the web [152].

2.8 User-Level File Systems

Deep Store is a user-level file system. Portability and long-term maintainability is a goal, not the very high performance that was once demanded by file systems that were tightly bound to the underlying operating systems. There are many examples of user-level file systems, including DAFS [97], GFS [51], and Swift [26, 25]. User-level file systems have advantages during development such as ease of programming and debugging, leverage of existing programming language libraries, and avoiding interdependence with an operating system kernel. The Inversion File System [115] used a database system to manage file data and provided features like database queries and file versioning; operations on files larger than 1 MB were about 70% of native Unix file system performance. These characteristics suggest that the use of a user-level file system would be suitable for our solution.

2.9 Archival Metadata and Searching

Moore *et al.* enumerate preservation and management policies that can be used to define the infrastructure for a collection-based archive [105, 106]. The problem domain for archivists is larger than simply to retain storage; however, the myriad requirements help define the usage for an archival system, including management through metadata, including metadata that use standardized formats to support versioning, security, and other desirable features in an archival storage system. Automated content type detection and metadata generation in systems like the Semantic File System (SFS) [146], Essence [58], and HAC [55] extract metadata from content itself, such as might be used by an archival storage system to annotate itself.

Search tools for file systems, such as GLIMPSE [101] and Google Desktop [1] extract metadata and index content, allowing users to search over file contents without reading the data directly. Index storage overhead is low. Connections [150] improves search in active file systems by incorporating temporal locality, which may also be useful for archival storage.

Earlier work to detect similar data in file systems by Burkhard *et al.* computed a distance metric in a key space [18]. It precomputes a set of keys for each file and organizes files into search spaces that can be traversed to find *cliques* of similar files. The large numbers of indexed keys produce large overhead, and the imposed organization of search data structures make this design less attractive for a distributed low-overhead storage system.

2.10 Reliability

Long term data permanence depends on reliable storage. Data storage reliability models are evaluated at different levels. Low-level storage depends on the storage medium technology, permanence of materials, and failure modes based on physical changes. Single-node reli-

ability storage models including RAID [34] to handle single or double failures [36] when data is stored using block-based device storage. Cluster-based reliability models like Ursa Minor allow the use of flexible reliability models on a per-object basis [2] that address the heterogeneous storage requirements of file system content. Xin *et al.* have demonstrated that two-way and three-way mirroring schemes optionally combined with RAID is sufficient to yield reliable large storage systems of petabyte scale [172]. Distributed file storage that distributes encoded file data, such as Rabin's Informational Dispersal Algorithm [128] and OceanStore [82] allow for reconstruction from m of n pieces, where $m < n$.

Chapter 3

PRESIDIO Overview

Our thesis is that a scalable archival storage system efficiently stores diverse data by progressively applying large-scale data compression in a single storage framework to provide better space efficiency than any existing storage compression method. The solution consists of the following: a system that identifies similar and identical data, methods to eliminate redundancy using one or more space-efficent storage methods, and a low-level content-addressable storage system into which data is recorded. The degree of similarity between new data requested for storage and previously stored data has a direct relationship on redundancy and therefore the compression rate. Depending on similarity, different types of compression algorithms are evaluated in a progressive manner to maximize storage benefit while minimizing computational and I/O cost.

We start this overview by presenting the Solution Overview, which describes the PRESIDIO progressive redundancy elimination solution and its relationship to the Deep Store architecture. Next, we describe a prototype implementation of PRESIDIO that was used to help validate results to support our thesis. And last, we give a brief overview of our evaluation of PRESIDIO and our main contributions.

3.1 Solution Overview

The PRESIDIO solution strategy is to apply progressively efficient data compression methods that meet performance criteria over large volumes of data, using a single content-addressable storage subsystem. The properties of data compression algorithms vary in many ways: the granularity of identifying redundant data, the computational complexity of detecting similar or identical data, the method to encode compressed data, and performance of the multiple stages of compression. Some compression schemes may yield high reduction of used storage with low space and time overhead, while others yield moderate efficiency beneifts with higher overhead, we employ an algorithm to progress through a number of compression schemes to find the most beneficial with the least cost.

Our data compression model expands on the generalized compression paradigm consisting of a *model* and a *coder*: the model predicts or defines the probability of the source, and the coder produces the code based on the probabilities. The application of this paradigm is present in most high-efficiency lossless stream encoders, as well as a part of lossy compression algorithms including JPEG, which incorporate lossless compression in a substage.

PRESIDIO differs from existing data compression models in the following ways. First, the data compression uses a much broader context, which includes all previously stored data within the archive. Second, because PRESIDIO is a hybrid compression algorithm, it depends on multiple algorithms whose probabilities of detecting redundant data differ. Third, codes that allow data reconstruction are very small compared to the large amounts of data that are stored, leading to high rates of compression when redundancy exists. Fourth, in contrast to many data compression formats, PRESIDIO virtual content-addressable (VCAS) storage is a non-linear format that supports hybrid methods for reconstruction. Finally, because our solution uses a hybrid compression scheme, the stages for compression and decompression are defined

by the PRESIDIO framework.

3.1.1 Large-Scale Data Compression

Our fundamental problem is compressing data on a large scale. Data compression is a large area of research that has matured in a number of areas such as lossless compression within streams, lossy compression for audio or video content, and compression of web/network or disk based content. Data compression algorithms are as diverse as types of applications and their output data. Data compression to reduce the redundancy existing in a large corpus is less mature, but information theory applies to it as it does to small-scale data.

The general model of data compression consists of a *model* and *coder* [111]. The model takes input data and rules that determine the codes to output. In *stream compression*, the input is a serialized data stream of symbols. Encoding the data in a more compact form reduces the space that is necessary to store the input data. Symbol encoding algorithms include *Huffman coding*, in which symbols are assigned codes based on static or dynamic probabilities, and *arithmetic coding*, in which probabilities of the occurrence of symbols are represented by subdivisions of a fractional numerical space. Frequently occurring symbols are represented by codes with fewer bits than infrequently occurring symbols.

Recent research and development in storage systems employ data compression [42, 125, 118, 40, 83, 163], but most literature published on data compression does not consider inter-file compression.

For the purpose of discussion, we define *inter-file compression* as the process used to eliminate redundancy across independently addressed objects. In addition, we use the term *large-scale data compression* to include both inter-file and intra-file data compression.

We can consider the two forms of compression we will consider in our solution as:

- *intra-file compression* consists of two stages, the *model* and *coder*
- *inter-file compression* consists of four stages, the *feature selection* and *resemblance detection* stages, followed by *redundancy elimination* and *recording to disk*

These two forms of data compression overlap in their operation. The model attempts to identify symbols that can be re-encoded, and then the coder encodes the data. Likewise, in inter-file compression, the *feature selection* and *resemblance detection* phases perform the same function as the model: they identify and then detect (large) symbols. Once these symbols are detected, then *redundancy elimination* will encode the data to eliminate duplication. The encoding process ends once the data is *recorded to disk*.

Each method can be used effectively for different types of redundancy. The long research history of stream compression has produced a wide range of space- and time-efficient algorithms that are effective at reducing stream sizes. These algorithms have trade-offs in compression and decompression speed as well as the rate of compression [170]. Likewise, inter-file compression performance—in all dimensions—is subject to trade-offs. In order to best compress data in a large-scale storage system, we use a combination of these methods to provide the highest reduction of storage with the lowest amount of space and time overhead.

Combining different forms of compression is useful. For instance, stream compression, which is restricted to examining and eliminating redundancy from the symbols seen within a stream, does nothing to eliminate copies of the same stream or file. Likewise, suppressing duplicate storage of files does not compress data within the files. When possible, we maximize complementary compression methods when it improves results.

The best strategy for efficiently storing different forms of compressed data is not directly evident. A wide variance in behavior, for instance, the average size of a stored file or object, or the amount of redundancy that exists in input files, can have significant impact

on the effectiveness of a storage system. When little or no redundancy exists in data, then overhead should be minimized; otherwise it makes little sense to use a "compression" scheme that increases storage usage. Likewise, if a storage system does not compress highly redundant data well for a particular application, then cost will increase and customized compression will be employed for that application. The development of a unified object storage mechanism will improve the ability for an efficient archival storage system to record data for a wide variety of data types and content.

3.1.2 Compressibility of Data

Data compression is data dependent, whether lossless or lossy. In lossless compression, the probabilities of the occurrence of characters, strings or other redundant data, depend on the type of content. Textual documents consist of many characters occurring with different probabilities, as well as words with non-uniform distribution and often as repeated substrings. More structured documents whose contents are defined by a file format, such as word processing documents, spreadsheets, and databases, use fixed fields, preambles, tables, and frequently occurring codes that bear no relation to natural language. Other documents, such as digitized analog data may consist of sequences of data that is noisy but may be compressed. Already-compressed files and files with random data have little redundancy and high entropy. These are some examples of reasons why there is high variability in compression rates of data written to archival storage.

The wide variation of data probabilities makes it more difficult to engineer a single efficient archival storage system to satisfy all types of input data. Our solution is to adapt using progressive compression methods, based on the content of existing data. Before proceeding to encode the input data, PRESIDIO uses several heuristic tests to determine the probability of

finding exact or similar data. When exact data is not found by using a fast hashing test, then another test is applied. If highly-similar data is not found using a hashing test with fingerprints, then another test is applied. And these tests are continued until they are exhausted or certain performance thresholds are exceeded. In this manner, probabilities for high similarity are progressively evaluated and discharged.

Once the tests for probability are evaluated, then coding (redundancy elimination and recoding to disk) is performed.

3.1.3 Efficient Encoding of Data

Encoding of data is the stage of data compression to reduce the stored data by retaining smaller codes to represent the uncompressed data. Common codes may be static codes like Huffman encodings, or they can be dynamically determined such that codes change their meaning in the compressed data stream. Designing space-efficient codes is important for the representation of data in lossless stream compression because they must be included in the output; their size matters significantly.

Our solution uses content-addressable storage with strings of data that range from hundreds of bytes to megabytes in size and are addressable by their content. *Content addresses*, which are of fixed size in the range of 16 to 20 bytes, are small enough such they can be used as references. A single content address can address an arbitrary amount of data. While content-addressable storage exists today that suppress storage of duplicates, our solution also encodes slightly different data with very low overhead.

The main themes employed within our solution are the following: using small content addresses, using low overhead to store data, and storing and reconstructing data indirectly through a virtual CAS.

3.1.4 The Virtual Content-Addressable Storage System

Within our solution for storing data efficiently is a virtual content-addressable storage (VCAS) system that addresses shortcomings in existing storage systems. File systems are primarily designed to provide high performance read and write operations rather than high space efficiency. For example, allocation blocks allow fast appends and updates, extents or sequential allocation of blocks improve streaming performance, and partial block allocation reduces fragmentation. In addition, directories store collections of files with rigid metadata that do not capture data properties that are necessary for archival storage.

Simple content-addressable storage, which stores full copies of instances, is not sufficient for storing multiple forms of compressed data, either. CAS space efficiency depends greatly on factors such as the average size of stored objects, the size of metadata—specified externally by users or applications as well as internally by the CAS itself, the type of compression methods being used, and most importantly, the level of redundancy in the data being stored.

The VCAS addresses the shortcomings of existing storage systems that arise when a wide variety of data is recorded to disk. It does this by presenting a single CAS programming interface, implementing a framework for storing and retrieving data polymorphically, and recording both small and large content-addressable data blocks into a unified CAS. The following description of the VCAS outlines the virtual data representation and then the storage operations.

3.1.4.1 Virtual Data Representation

The VCAS stores data virtually in a manner that is transparent to the larger archival storage system. By doing so, the VCAS appears to be a standard CAS, but internally it stores the content-addressable data objects more efficiently. This design solves a number of problems.

First, objects stored in the VCAS are addressed by content, as in a regular CAS. Addresses are computed by hashing the contents of the object being stored with a high probability of uniqueness.

Second, objects are stored with little overhead. This permits the CAS to be used with small objects. In some applications, small files will be stored. In other instances, efficient storage methods in PRESIDIO may store sections or encodings that are smaller than the average file. Low overhead also allows storing completely unique data without incurring the overhead a regular CAS or file system would impose.

Third, the content address (CA) data type is small, on the order of 16 or 20 bytes, but similar in size to other CAS systems. Because the CAs are used internally and externally, the address size contributes to the storage overhead.

Fourth, objects are stored virtually. The storage and reconstruction processes use simple rules to construct arbitrarily complex internal representations of the stored data. Virtual storage is flexible enough to provide literal storage, *i.e.* traditional CAS operations where single, raw or stream-compressed instances of objects are stored in their entirety, as well as subdivided sub-file chunks or delta-encoded storage. Content addresses are embedded within the virtual representations; the VCAS interprets the virtual objects and the content addresses during storage and retrieval operations.During a storage request, the VCAS converts input, or concrete representations, into virtual CAS objects. During a retrieval request, the VCAS converts the virtual representation back into the concrete representation. The VCAS offers other desirable properties: the content address is used to ensure the integrity of the object, and the polymorphic behavior allows the VCAS to be extensible to use other internal virtual representations.

Last, internal and external object metadata are stored using the same CAS as the object content metadata. File metadata are stored as VCAS objects, as are internal data to help

the large-scale data compression methods.

3.1.4.2 VCAS Storage Operations

Our VCAS subsystem presents a simple external storage interface. The operations are designed with a number of criteria in mind. They should be *simple*, so that specification, implementation, verification, and accessibility far in the future are possible. They should be *flexible*, to allow a variety of data and file formats to be stored and without limitations such as those placed on metadata like filenames or on structure like singletons of flat files. And they should be *complete*, to provide a round-trip guarantee that all data can be stored and that all retrieved data is identical to the stored data.

The storage interface consists of operations on objects, such as files or file metadata, that are addressed by content. The storage operations are:

- Store object
- Retrieve object

This simplicity in design is motivated by the need for long-term preservation. For content-addressable storage of disparate types using different efficient storage methods, we required a uniform storage interface. Toward the goal of long-term preservation, a simpler specification and ease of implementation helps ensure that data written today from one client system can still be read from a completely different client system in one, ten, or even a hundred years. These operations form the basis for other storage and retrieval operations like storing files with metadata.

Retrieving data from an archival store can be difficult if the encoding of stored data requires complex interpretation, or if instructions are not self-evident or explicitly defined. Our

solution aims to outlive changes in computing and storage technology by combining a simple file storage and identification interface with an efficient storage mechanism that separates the mechanisms to detect and eliminate redundancy from the data format and specification to retrieve and reconstruct files from their component parts. An example of how even well-intentioned archival digital data is fragile is evident in the attempt to recover and make accessible data from the Domesday Project of 1986, in which British life at that time was chronicled to complement the original paper Domesday Book (*c.* 1086) [48, 28]. The multimedia presentation was recorded onto LVROM videodisc media designed for permanence of about 100 years. Unfortunately, wear on the media and the obsolescence of the LVROM players shortened the useful life of this digital artifact. More importantly, the software to present the materials was not portable, limiting interpretation of the data itself. Consequently, the preservation effort aims to emulate the combined hardware and software of the original Domesday system. Although it is not the intent of our project to solve the problem of interpreting application content, our solution aims to ensure that the data stored internally is easily interpreted and can migrate to future systems.

The VCAS storage interface is intentionally designed not to meet certain non-goals. Application- or user-defined inter-file relationships are not specified explicitly by any operation; it is the responsibility of the storing application, which includes the filing programs that are the analog of file systems, which provide directories and linkage, to store the relationships. Internally, the VCAS creates inter-file relationships in order to reduce storage of redundant data, but these relationships are not made visible to the client of the interface.

The storage operations and interface are described in more detail in Section 7.4.

3.1.5 PRESIDIO and the Deep Store Architecture

The Deep Store architecture consists of these primary abstractions: storage objects, physical storage components, a software architecture, and a storage interface. We briefly describe each of these in turn.

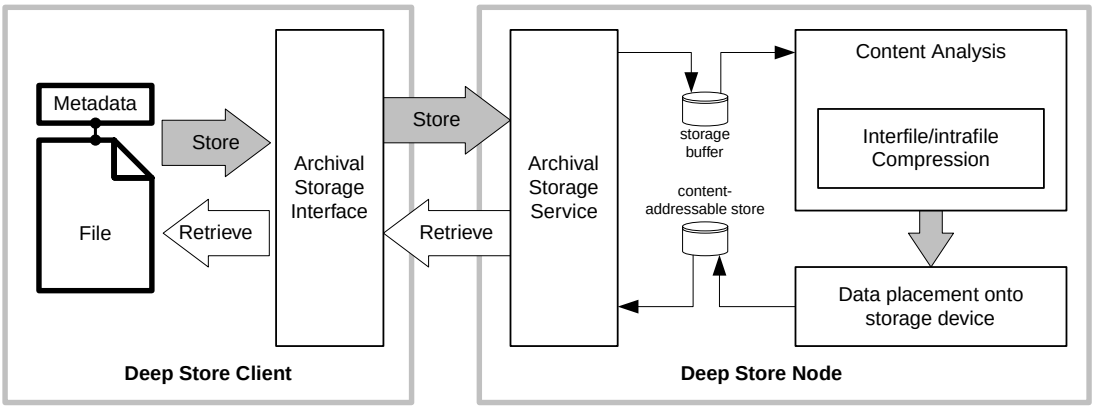


Figure 3.1: The Deep Store archival storage system model

The primary storage objects presented to the archival store are the *file* and its *metadata*. A file is a single contiguous stream of binary data. It is identified by its content. A hash function, such as MD5 [133] or SHA-1 [114] digest, is computed over each file to produce the primary portion of its *content address*. File contents are stored as content-addressable objects, devoid of any metadata. Simple metadata associated with the file, such a file's name, length, and content address, are contained within a metadata structure. The metadata can also be identified by content address. Content-addressable storage (CAS) systems like EMC Centera store files in this manner [42]. Section 7.3 discusses the use of file metadata in detail.

The primary unit for storage is a Deep Store *storage node*. Multiple nodes connect over a low-latency/high-bandwidth network to make up a *storage cluster*. Each node contains a processor, memory, and low-cost disk storage. Content analysis, which is absent from most

file systems, includes fingerprinting, compression, and data storage and retrieval; these are necessary operations, but they must have high throughput to be practical.

While the additional processing of compressing data might reduce throughput, the reduction in stored data can also make a positive contribution to performance. Content analysis is performed on silicon, and the performance growth rates for silicon-based processors and memories historically have been greater than the capacity and latency improvements of magnetic disk. Considering the ever increasing CPU-I/O gap, this disparity implies a potential performance benefit to effective bandwidth from the reduction of stored content—in other words, compression can benefit I/O performance.

The software architecture is embodied in processes that execute on each storage node: an *archival storage service*, a temporary *storage buffer*, a *content analyzer*, and a *content-addressable store*. The archival storage interface accepts input using a common system interface such as function interface, pipes, sockets, or WebDAV, all of which we have found to be suitable for a simple client-server request mechanism. The storage buffer minimizes request latency; our implementation stores the uncompressed content in a CAS. The content analyzer eliminates redundancy and can also be used for extracting metadata. The efficient CAS stores content and metadata alike.

3.1.6 PRESIDIO Hybrid Compression Framework

PRESIDIO is designed as a framework defining a storage model, efficient storage methods, and an algorithm to combine the two. The heart of this storage system is the CAS. Efficient storage methods are object-oriented classes that implement several operations on a single piece of data: feature selection, resemblance detection, and redundancy elimination. The storage methods also perform the two parts to write and then read the data. To record the data,

a storage object is represented as a virtual object; the virtual object describes itself as a series of actions to encode itself using the CAS. To reconstruct the data, the self-describing virtual object describes the steps to reconstruction. The instructions are defined as concatenations of other stored CAS objects, delta-compressed CAS objects, stream-compressed data, or just raw bytes.

To determine the most desirable efficient storage method, the PRE algorithm iteratively applies feature selection and resemblance detection to a candidate file to compute a ranking based on the estimated storage efficiency. Once all ranking has been completed, the most efficient method is used to record the object. Because the most efficient storage methods vary based on the content and estimated storage efficiency, more than one method can be applied to a file, thus creating a hybrid compression storage mechanism within the PRESIDIO framework.

3.2 Prototype Implementation

We have implemented a prototype of PRESIDIO that demonstrates the ability of our solution to eliminate redundancy progressively within an experimental VCAS and efficient storage method framework.

3.2.1 Feature Selection

Feature selection is performed using a variety of data fingerprinting methods. Feature sizes range from tens of bytes, in the case of shingling over sliding windows, to entire files, in the case of whole-file hashing. We have implemented a flexible Rabin fingerprinting library in C++ for this purpose. It has been improved for sliding windows on byte boundaries and feature selection over sets of fingerprints to make it perform close to the bandwidth of disks. Appendix A provides implementation details and some technical details we found important to

our work.

Other feature selection methods like whole-file digests are readily available. We selected pre-built executables like *md5sum* or libraries like *libopenssl* for their availability and conformance to specifications.

3.2.2 Similarity Detection

We experimented with similarity detection first using standalone executable programs such as *chc* (Chunk Compression) and *dcsf* (Delta Compression between Similar Files) to determine their efficacy, then later bringing the similarity detection algorithms into the PRESIDIO application.

The hash-based storage and retrieval was implemented using the *Berkeley DB Database* [149, 144], a multipurpose embedded database storing key-value pairs in linear *hash* and *recno* (record number) databases. Similarity detection by matching fingerprints and superfingerprints was also performed by storing feature sets, or *sketches*, and using inverted term lists to find similar data.

3.2.3 Virtual CAS

The Virtual Content-Addressable Store was implemented as a C++ framework with multiple object-recording implementations: a file-based CAS in which VCAS objects were stored individually as files in a directory, a Berkeley DB *hash* database in which objects were stored as *values*, a Berkeley DB *recno* in which objects were also stored as *values*, and a hybrid Berkeley DB *hash* database mapping Content Addresses to *recno* entries in a second database, and a large-block file storage.

The polymorphic design of the VCAS allows different implementations to be used

simultaneously for different purposes. For instance, the file-based CAS is sufficient for small numbers of files when used in the *storage buffer* shown in Figure 3.1, but the hybrid CAS is used to meet the requirements of storing large numbers of files in the *content-addressable store*, also shown in the same figure.

3.2.4 Redundancy Elimination Framework

The Efficient Storage Method (ESM) framework is the other major component to PRESIDIO. Within it, multiple methods such as whole-file hashing, chunking, and delta compression between similar file are applied to data to compress it. More than one ESM may be used to detect and then to eliminate redundancy. A second part of this ESM framework is the selection algorithm. The PRESIDIO algorithm chooses the best method based on the data, performance constraint, and probability of redundancy. Combined together, these parts assemble to form the PRESIDIO CAS.

Chapter 4

Identifying Data

In this chapter we describe the methods for identifying data using *features*, *data fingerprinting*, and *feature selection*. These content analysis tools are used to help identify similar and identical data within a large corpus. The organization of this chapter is as follows. First, an overview defines a number of terms used in data fingerprinting and the feature selection process. Second, we describe what features are, followed by a survey and brief analysis of hashing and fingerprinting algorithms we use in our solution. Third, we enumerate and detail the feature selection algorithms we use. Fourth, we present the main feature selection methods used within our solution. We close with a summary of feature selection.

4.1 Overview

Efficient archival storage requires tools to help identify and then eliminate redundant data. Identification of similar and identical data relies on hashing, fingerprinting, and digest algorithms that make it possible to deterministically compute an identification value for a range of data into a much smaller fingerprint.

We start with a few definitions to help unify concepts used within our storage compression framework. We define *feature data* as a substring of data—a contiguous sequence of bytes of non-zero length. It may be a proper substring or the entire contents of a file. Each *feature data region*, the starting and ending positions bounding the feature string data, may overlap with other feature data regions, or they may be distinct. *Storage objects* include files, variable- or fixed-sized blocks from a file, or other content including metadata. From these objects, we compute features to identify them. *Features* are fingerprints of feature data. *Fingerprinting* computes a hash, or fingerprint, over a feature string with the usual hashing property of providing a much smaller name for an object with low probabilities of name collision. A collection of features (fingerprints) forms a *feature set*. Operations on feature sets are desirable because they can compute approximate similarity metrics between files that are highly correlated with similarity operations between files without re-reading contents of files, dramatically reducing computation. This heuristic approach works well over large data sets when direct comparison would be infeasible. *Feature selection*, also known as *feature extraction*, is the process of determining a characteristic set of features. An underlying theme in our experiments has been the wide variety of data that are stored as archival data, but finding a single type of feature that can apply to all data is challenging. Large numbers of smaller features and content-dependent features such as text extraction improve resemblance detection, but they can increase storage overhead or may not apply to other classes of data.

Figure 4.1 illustrates the relationships between the algorithms and objects that are produced during the feature selection process. The foundation includes simple bit-string hashing and data fingerprinting algorithms. Variable-length subsections of files can also be used as content from which to compute features. Shingling methods compute fingerprints from overlapping (sliding) windows over content.

The bottom row lists the main features we use: digest, chunk digest, chunk lists, sketches, and superfingerprints. Digests are hashes computed over arbitrary binary strings such as whole files. Chunk digests are the same, but computed over chunks, or subsections, of files. Chunk lists are concatenated identifiers to chunk data; the identifiers are usually chunk digests. Sketches are summary collections of identifying data, like data fingerprints, selected deterministically from a larger set of fingerprints; in addition to identifying data, they can be used to compute a similarity metric. Sketches that are serialized into a stream of data can also be used as input to a fingerprinting function to compute superfingerprints.

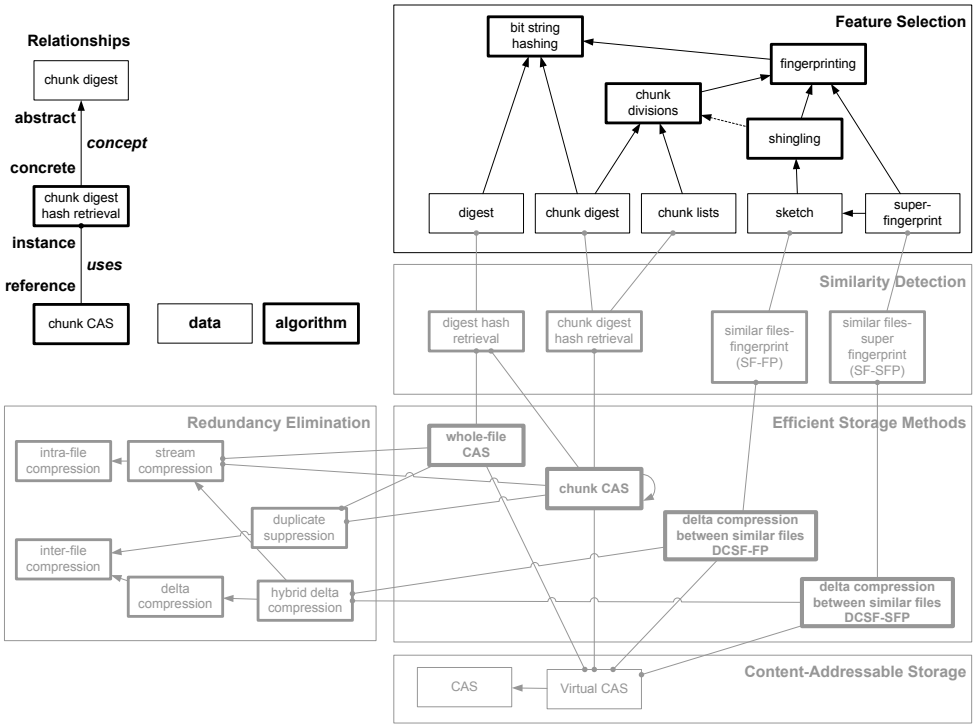


Figure 4.1: Fingerprinting methods used for feature selection

4.2 Features

Features are properties that identify and distinguish data. We differentiate between *feature data*, the original binary data, and *(computed) features*, the features computed from the feature data. Developers arbitrarily design features for their purpose, and usually based on the content and by design constraints. In our work we have chosen to use the least constrained, most general content-independent algorithms operating over arbitrary binary data when possible so as to apply features to a wide range range of problems.

When the format and contents of data are known *a priori*, features that yield higher quality metrics are possible. For example, information retrieval and web search systems use human-readable words extracted from content to determine features made up of words, word bases, sentences, word sequences, as well as numerical or hash representations for those words. Metadata, both manually or automatically generated, are also used as features. These include tagged data, filenames, keywords, and time stamps.

If such a wide range of metadata and feature types are available, why do we not use them? Semantic feature computation and selection algorithms that require knowledge of the data encoding have several shortcomings for the purpose of storing data efficiently. First, the content types must be well-specified. Data encoding standards exist, but they change or update relatively quickly, making it difficult to predict what features used today will be used in the future [135]. A striking example is the common conception that PDF (Portable Document Format) is suitable for archival use; however, work on the archival PDF/A standard is only in progress now [85, 119, 74, 73]. Second, interpreting content implies ensuring algorithmic permanence in addition to data permanence—in other words, semantic interpretation of the data must be possible well into the future when data is read, and not at present when it is written. This is a difficult problem that is well outside of the domain of our problem [91, 92]. Third, searching

over the stored data is a different problem than identifying data for data compression. Current indexing and retrieval techniques differ based on content type. For example, text searches that use inverted file indexing [170] or PageRank functions [10, 9] require low latency—not high throughput or space efficiency—and serve the purpose of searching for arbitrary user queries.

The goals for storing data efficiently and searching semantic content are different, but not mutually exclusive. Instead of incorporating them into a single system, we defer the design of content-sensitive selection and search mechanisms to higher levels of the system and focus our solution on the problem of storing data efficiently. We spend the rest of this chapter describing content-independent feature selection algorithms.

Figure 4.2 illustrates some examples of different types of feature data that can be found in files. File content and length are shown as long horizontal rectangles. The shorter rectangles—contiguous regions of data within files—represent substrings of binary data, which represent individual *feature data*. Operations on feature data such as testing for equality are unwieldy so strategies for identifying the data in a more compact form is essential for improving performance. The primary type of feature we use is fingerprints, which are hashes computed from the features that are the file contents. Fingerprints, which are small and of fixed size, easily identify large amounts of data uniquely enough.

The meaning of features may be implied by context. For example, fingerprints used in the *rsync* program to determine substrings can be considered features, but they exist within a context, namely the comparison between two files that are known to have an existing relationship. We will discuss the differences across these features in more detail later this chapter.

Features may also be inferred. For instance, data clusters reveal certain features that may only exist when they are in collections. Machine learning algorithms may also extract features from serial data that is analyzed over time.

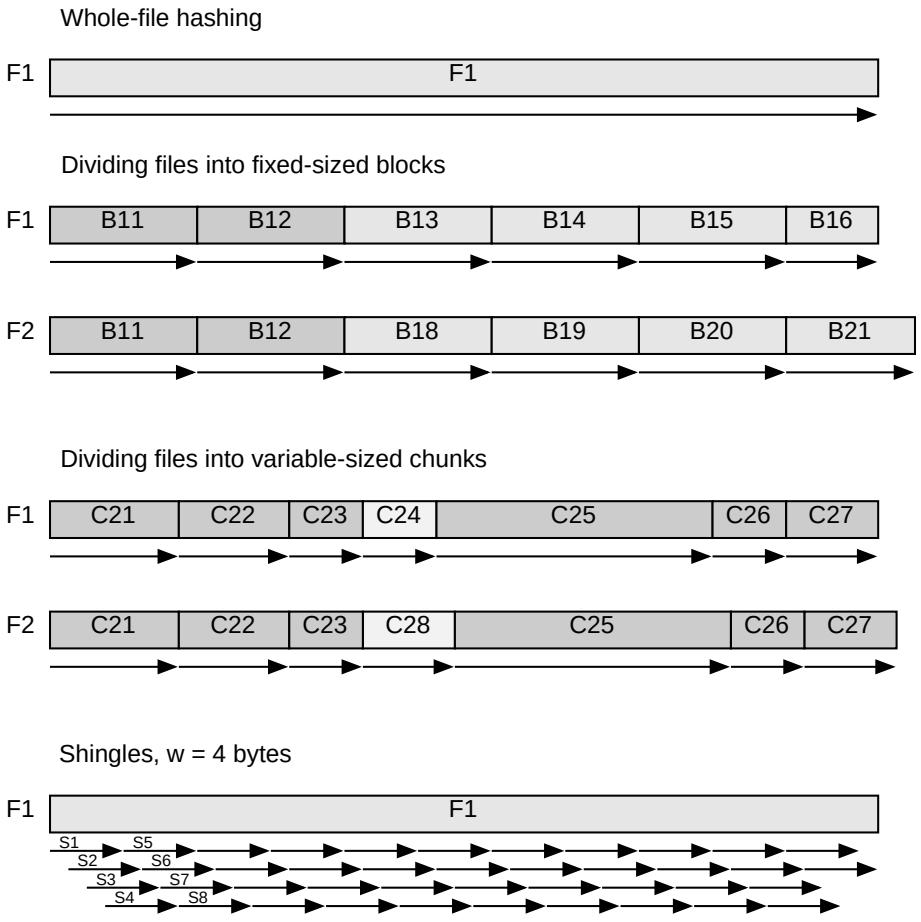


Figure 4.2: Examples of file features

In order to minimize dependencies across a very large system, our design uses strictly deterministic feature computation and selection, *i.e.* computing and selecting features is purely algorithmic and not dependent on data already stored in the system. Independence across a distributed system allows it to operate without interaction, thereby improving the opportunities for scalability. Feature selection that is independent of existing data may offer performance benefit within a single storage machine by avoiding extra I/O due to data retrieval before data storage.

4.3 Hashing and Fingerprinting

Two of the most important tools we use for identifying content are hash tables and hashing functions. In order for our storage system to manage large volumes of data in the range of petabytes to exabytes, we first use unique identifiers based on content and then store them using hashing data structures. This content-based addressing strategy meets the following deisgn requirements:

- identifying data deterministically with no prior knowledge
- high probability of identifying identical content
- low probability of mis-identifying non-identical content (false positives)
- low computational complexity finding similar data in a storage system
- computing data identifiers is efficient

Hashing is the method we use to represent arbitrarily large strings of binary data as smaller strings that meet these properties. An arbitrarily large sequence of n bits is concatenated to form a *bit string*, equivalent to a *byte string* of $n/8$ bytes.

A hash function maps the space of bit strings into the space of hash values. The space of all possible permutations of that bit string is 2^n. A hash function produces *hashes* or *hash identifiers*, which are much smaller bit strings. Our solution uses hashes of fixed length m, whose value is typically tens to hundreds of bytes. Thus, the hash function maps the space of permutations 2^n into 2^m possible values. Because $m \ll n$ not all possible bit strings can be represented unambiguously. But when m is 128 or larger, the range of 2^m hash values is large enough to uniquely identify data with high probability. Hash functions are designed to avoid *collisions*—duplicate hash values for different inputs—with any change in input.

We seek to produce evenly distributed hash results regardless of the input. For a fixed fingerprint size, Rabin fingerprinting by random polynomials defines easily tested criteria (irreducible polynomials) for creating functions for the size. For each size, the set of Rabin fingerprinting functions forms a *universal* class of functions, from which a function may be selected at random. Carter and Wegman have shown how random selection in *universal hashing* avoids biased hash results [31, 161].

Broder narrows the definition of *fingerprinting* [11] from the more general *universal hashing*. In (universal) hashing, n, the number of distinct objects, is a fraction of the total number of possible fingerprints 2^k, whereas in fingerprinting, $n \ll 2^k$. In other words, fingerprinting relies on the low probability of collisions to achieve near-uniqueness, whereas general hashing often assumes hash collision is common enough to design hash tables around collision resolution algorithms such as *open hashing*.

Maximizing I/O performance in storage systems is always a goal. Modern computers and storage systems bypass the CPU by using *direct memory access* (DMA); data is transferred directly between primary memory (RAM) and secondary storage (disk). When we introduce content analysis like fingerprinting into the storage pipeline, storage bandwidth is reduced, so it is important to use hashing functions with very low computation complexity.

4.3.1 Collision Properties

We rely on low collision rates for identifying data; when evaluating content-addressable storage for identifying blocks with probabilistic uniqueness, it is important to consider how it affects the design of the storage system. We discuss content address size and collision rates in detail in Section 7.2.1.1.

4.3.2 Digests

Functions that accept variable-length strings are often called *message digests* to reflect the summarizing effect of the hashing process. Digest functions include *Rabin fingerprinting by random polynomials*, string hashing in the *Karp-Rabin* string matching algorithm [76], Pearson's *Fast Hashing of Variable-Length Text Strings* [120], and cryptographic hash functions like MD4 [132], MD5 [133], SHA-1 [113], SHA-256, SHA-384, and SHA-512 [114].

Digest functions hash the contents of files into small *digests* in the range of tens of bytes. Most digest functions that are used today for fingerprinting large blocks of data are *one-way hashing functions* or *cryptographic hashing functions*. The main reasons for this practice appear to be consistency, availability, and performance, and not necessarily their security properties. Digests must be well-specified for cryptanalysis, and implemented and deployed widely. And because cryptographic functions often are used in high-bandwidth applications, they must exhibit high throughput.

4.3.3 Rabin fingerprinting

Rabin fingerprinting by random polynomials is useful for hashing and fingerprinting. It has been popular with high-bandwidth networking and disk-storage applications for many reasons [11, 127]. The most important properties that apply to our solution are

- efficient computation of fingerprints over sliding windows
- improved performance by precomputing tables for byte- and word-level operations
- availability of a large number of randomized functions that can be used for feature selection

Broder identified other desirable properties such as the ability to compute fingerprints over two concatenated strings as a function of the fingerprints of the two strings. This permits parallel computation of subsections of a large file, unlike some cryptographic functions like MD5 and SHA-1, which compute chained results.

There are a few shortcomings to using Rabin fingerprinting, none of which significantly affect the quality of the fingerprints when they are used over random data. See Appendix A for details. However, from an implementation standpoint, there are no existing specifications so at best *de facto* standards are established by implementation.

4.3.3.1 Toward a Specification

We have found that the lack of specification revealed differences in several implementations, including those used in the Low-Bandwidth File System (LBFS) [110] and the Microsoft Research implementation [99]. At least these parameters should be specified:

Conventions required in a Rabin fingerprint specification

- size of the polynomial and fingerprint
- bit ordering of polynomial and fingerprint
- byte ordering of polynomial and fingerprint
- bit ordering (which bit is highest coefficient within a byte) of data
- byte ordering (which byte is highest within a word) of data

Parameters in a Rabin fingerprinting function that will change hashing properties

- irreducible polynomial (*i.e.* polynomial coefficients)

- degree of the polynomial
- fingerprint for the "empty string" (or equivalently, a bit string prefix for all strings)

Conventions used in our application of Rabin fingerprinting During our implementation, we observed the effects of these conventions and parameters on the functions. We selected the following conventions; Appendix A provides a more detailed description.

- polynomials of degree 32 and 64, and fingerprint representations of 32 and 64 bits, respectively
- all polynomials of degree 32 were monic in the highest degree; in other words, a polynomial of degree 32 would have a high-order coefficient of one, resulting in $x^{32} + \cdots$
- big endian bit and byte ordering of data to the string to be fingerprinted

  Our choice to use big endian was somewhat arbitrary, but we decided it was easier to understand the implementation by observing bit values stored in bytes so that low-order memory represented high-order polynomial coefficients.
- big endian bit and byte ordering of the irreducible polynomial used in the fingerprinting function and also in the fingerprint itself; we use the same convention for both cases

The irreducible polynomials used in the fingerprinting must be selected once and used for all data. To do this, we wrote a program to test for irreducibility described by Rabin [126] using an pseudo-code algorithm from Chapter 2 of the Handbook of Applied Cryptography [102]. To ensure a high degree of randomness, we first generated a 32-bit (or 64-bit) pseudo-random number to represent the lower 32 coefficients of the polynomial, and then tested the polynomial for irreducibility.

4.3.4 Strong and Weak Hashing

We use hash functions in two distinct ways: *strong hashing* to uniquely identify variable length data across the storage system, and *weak hashing* to identify smaller strings using smaller hash identifiers with lower probability of uniqueness. We adapt the notion of strong and weak hashing from Tridgell's PhD thesis on *rsync* [164].

Strong hashing is used to identify large blocks of data, including whole files, to produce an identifier that uniquely identifies that object by its content. We define the computed hash of the content as its *content address*. Hash identifiers are represented as bit strings large enough to be probabilistically unique over all the data. Identifier sizes of at least 128 bits (16 bytes) are used for this purpose. The advantages to strong hashing are the high speed of many implementations, but more importantly allow exact duplicate data to be detected with extremely high probability and non-matching data to be detected positively.

In addition, *cryptographic* or *one-way hash functions*, which make undetectable modification of content difficult and that also have high throughput, are attractive algorithms for improving the tamper-resistance of immutable data.

Weak hashing in the form of fingerprinting is used to identify smaller regions for many reasons. These hashing regions are smaller than stored objects, and identify data with the assumption that small contiguous regions are more likely also to be present in similar files than not. We will describe how fingerprints over smaller regions are used to identify similar data. Our solutions use weak hashing in the following ways:

Shingling An algorithm to compute fingerprints over small, overlapping regions in a file.

Chunking A technique to subdivide files deterministically so that two independent processes produce identical subfile boundaries for identical files and nearly identical division points

	weak hashing	strong hashing
collision resistance	yes	yes
pre-image resistance	no	yes
weak collision resistance	no	yes
efficient over sliding windows	yes	no
randomized hashing functions	yes	no

Table 4.1: Requirements for weak and strong hashing properties

for similar files.

Feature selection A method to select a set of features (often fingerprints over shingles) using a randomized algorithm.

Superfingerprints A fingerprint over multiple fingerprints.

4.3.4.1 Useful Hashing Properties for Feature Selection

We make the following distinctions between these two forms of hashing, as shown in Table 4.1

If an input string is A, the hashing function h computes the hash value a: $a = h(A)$. Hash functions always determine inequality, so if $h(A) \neq h(B)$ then $A \neq B$. The opposite is not true of hash functions, however the required *collision resistance* property is satisfied if the probability $h(A) = h(B)$ is very small if $A \neq B$.

The *pre-image resistance* property is required of one-way and cryptographic hash functions: if the hash value a is known, then it is computationally difficult to find the input string A. Historically, it has been difficult to ensure this property over long periods of time as computing power have enabled brute force attacks and cryptanalysis have rendered many digest functions as flawed or less secure than anticipated.

Another cryptographic property, *weak collision resistance*, is satisfied when it is com-

putationally difficult to determine a different input which will hash to the same value, *i.e.* it is difficult to find A' such that $h(A) = h(A')$. It has recently been shown that this property is more vulnerable in functions such as MD4, MD5, HAVAL-128 and RIPE-MD in which hash collisions were discovered [166]. However, this property remains important to archival storage—probably more so than pre-image resistance—and guards against tampering of immutable data by assuring that once data is recorded and then a strong hash is computed, it is difficult to modify the data without invalidating the hash.

There are also properties that distinguish weak and strong hash functions in practice. Computing hash values must be *efficient over sliding windows* to provide high throughput when analyzing large volumes of data. Cryptographic hashes often use minimum block sizes and pad small strings, which are typically the case of sliding windows, thereby dramatically reducing efficiency. Also due to their cryptographic nature, they are unable to update a hash dynamically by introducing data entering the window and removing the effect of the data leaving the window. In contrast, some hash functions like Rabin fingerprinting by random polynomials will incrementally compute hashes over sliding windows with low overhead.

Finally, the *randomized hash functions* property can be used for feature selection. To select a randomized hash function, first a hash function is selected at random once and then fixed thereafter. In the case of Rabin fingerprinting, random polynomials are generated and then the polynomials are tested for irreducibility. The result is a new hashing function that will emit a different set of hash values for a set of input than another randomized fingerprint function using a different polynomial. The randomization improves the probability that no two functions will produce similar results over any set of input.

4.4 Feature Selection

Feature selection is the first major step for identifying identical similar data and for naming objects for content-addressable storage. In a manner similar to the way data compression algorithms identify strings and tokens, feature selection defines the rules for feature strings that are fingerprinted into features, and the algorithm for selecting the features to be retained.

In the following sections, we describe and analyze several feature selection algorithms for both cases of identifying identical and similar data.

4.4.1 Common Properties of Features

A stored object, whether it is a file or a sub-file region, can contain an arbitrary number of feature strings. Feature selection defines two things: the feature strings to be fingerprinted, and the *feature selection algorithm* for retaining fingerprints. In some cases, the feature set may be small; for instance a file digest would compute one feature over a feature string that is the whole file. In other cases, the number of features that are computed is very large, as is the case in shingling. Retaining all computed features is not always practical, and as described below, is not necessary for the purpose of approximate similarity metrics; therefore the selection algorithm plays an important role in determining feature set quality.

A feature set may contain multiple instances of features. An instance when this would occur is when randomized functions select two features representing the same feature data twice. We sometimes speak of features as both the feature string, as well as the feature fingerprint. The literature mixes these terms also, for instance "shingles" sometimes refers to the literal substring and at other times the fingerprint of the shingle.

Feature sets are stored as a representation of stored objects. There are several reasons. The first is that we assume immutable data, so once the features are computed, there is no need

to update them. The second is that computing and selecting features can be I/O and CPU intensive. The third is that fingerprints are much smaller than the data they represent, so the additional storage overhead is small.

4.4.2 Whole-File Hashing (Message Digests)

Whole-file hashing computes a hash value by using the entire contents of a file as the feature string to produce a single feature. It is identical to computing a *message digest*. The hash function can invertible or cryptographic (non-invertible). Invertible functions, like Rabin fingerprinting by random polynomials or the Karp-Rabin hashing function, make it very easy to tamper with the contents of objects while preserving the hash value. In contrast, cryptographic hash functions are designed to make tampering much more costly, if at all possible. Hash functions can produce variable sized hash values, but usually they are designed to produce fixed size values.

Within archival storage, the main goal of whole-file hashing is for storage clients and servers to produce a *content address*, an identifying machine-readable name for a file. Using whole-file hashing conveniently solves several problems: immutable file data is identifiable by content address; content addresses are very small (for instance 16–20 bytes) to reduce identification metadata; and hashing can be used to locate a file object in a flat namespace instead of traversing directories or other search domains, suppressing storage, and minimizing or eliminating data transfer by using content addresses instead of file file contents.

Whole-file hashing still exhibits several problems. The first is that file data must be immutable, otherwise the hash is invalidated. Second, hash functions do produce colliding results, making it possible for a file not to be stored because the write operation was suppressed on account of the apparent existence of the file's hash value for a different file [60]. Third,

hashing requires computation that may affect performance.

Furthermore, cryptographic hash functions may not stand the test of time. Many functions have shown weaknesses over time either due to analysis that reveals problems, or because computing power in the future will allow brute-force attacks on algorithms designed in the past. Archival storage systems that assume today's hash functions are secure may become vulnerable eventually.

4.4.3 Chunk Features

Chunking subdivides files into variable-sized blocks. Chunking divides files deterministically into variable-sized chunks of which single instances are stored. The technique first used in the low-bandwidth file system (LBFS) [110] divides files deterministically into chunks and then identifies chunks by hashing their contents.

We use two algorithms for chunking to solve two problems. First, a data stream, such as a file, needs to be divided into chunks in a deterministic way. We consider the general case of variable-sized chunks, which works for any type of data, including binary formats. Second, the feature content—data within a single chunk—is used as input to compute chunk features.

Dividing Files Into Chunks Chunk boundaries are defined by calculating some feature (a digital signature) over a sliding window of fixed size. In practice, we use Rabin fingerprints [127], for their computational efficiency in the above scenario. In Figure 4.3 the file stream is represented by the longer horizontal rectangles, and chunk boundaries are indicated by the short vertical lines. Boundaries are set where the value of the feature meets certain criteria, such as when the value, modulo some specified integer divisor, is zero; the divisor affects the average chunk size. Such deterministic algorithms do not require any knowledge of other files in the

	chunking parameter
w	constant-sized window
m	minimum chunk size
M	maximum chunk size
d	expected chunk size
r	residue (fingerprint mod divisor)
RF_{size}	degree of Rabin fingerprint (bits)
RF_{poly}	random irreducible polynomial
CID_{size}	size of chunk ID hash (bits)

Table 4.2: Chunking parameters

system. Moreover, chunking can be performed in a decentralized fashion, even on the clients of the system.

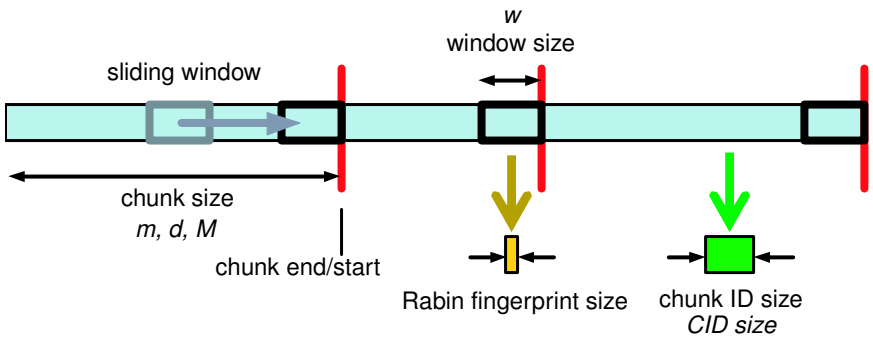


Figure 4.3: A graphical representation of chunking parameters

Chunk size distribution depends on several factors, but most notably d, the divisor. Typical distribution of chunk sizes varies whether the content is structured content, such as binary content in PDF files, or if the data is purely random. We evaluated two different data sets: a single file, 100 MB (100×2^{20}) containing random bytes; and a *tar* file containing mostly unique PDF files, total size 238.7 MB (754 files with the following size statistics: mean 324.1 KB, median 124.4 KB, standard deviation 783.2 KB). Parameters m = 64, M = 16,384, d = 1,024 bytes.

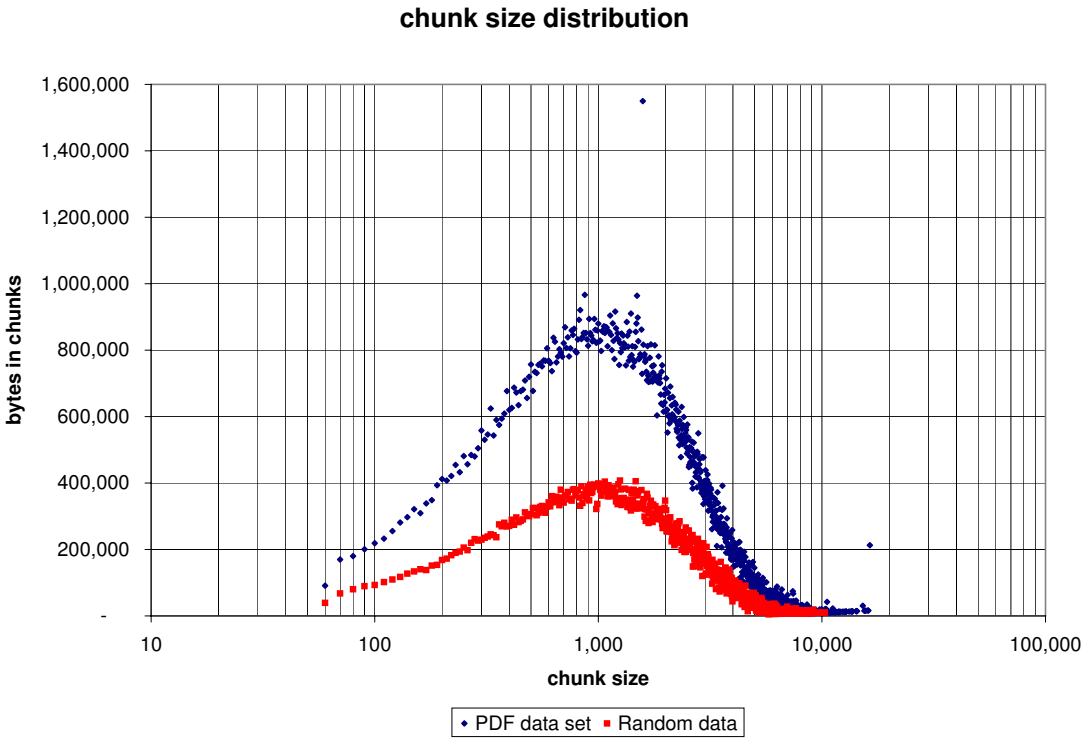


Figure 4.4: Chunk size distribution

Figure 4.4 is a histogram showing the distribution of chunks weighted by their sizes (horizontal axis) and the total number of bytes stored for chunks of that size (vertical axis). The distribution is representative of a wide variety of file types. The sum of the height of all points is equal to the total number of input data.

Two data points in the PDF group are particularly interesting. The first is the point located where chunksize equals 1,580 and the bytes in chunks stored is 1,549,766. This point represents a highly replicated chunk whose size is within the interval [1580, 1589] or approximately double the number of adjacent chunks of similar size. This tends to indicate a particular pattern that is commonly written in the data stream, either part of the tar file format or within

the PDF file format itself. The other data point is at M, or 16,384 bytes, a sum of the tail distribution.

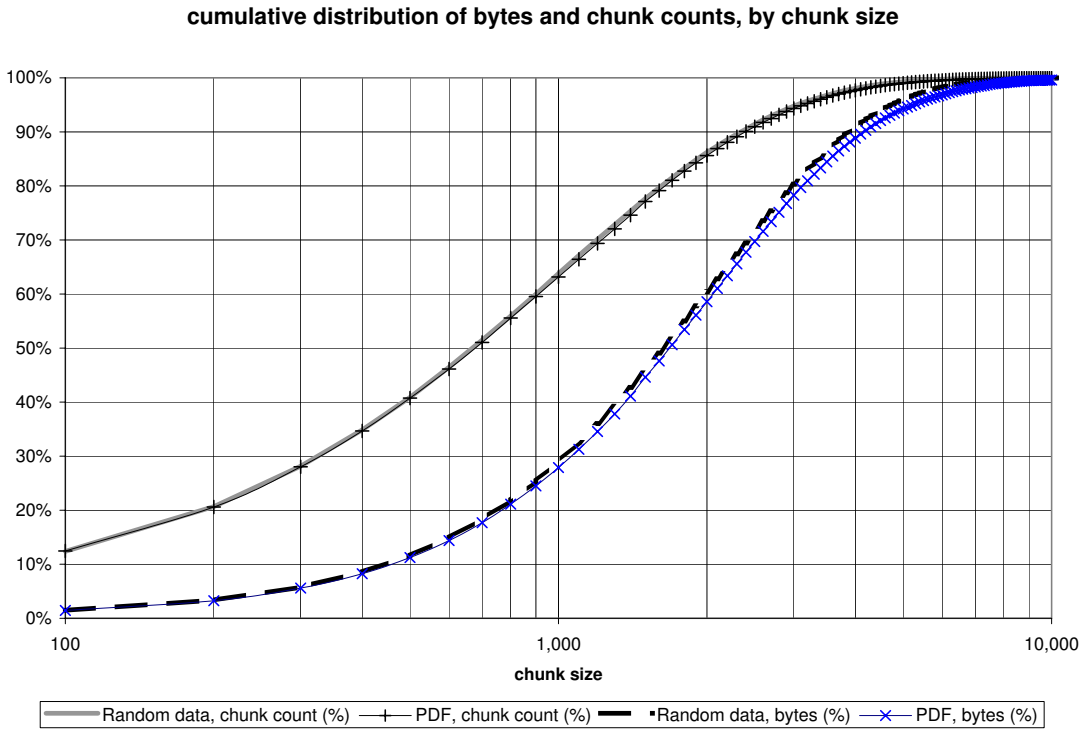


Figure 4.5: Cumulative chunk size distribution

Cumulative distributions of chunks by their sizes and counts and by their sizes and total storage are shown in Figure 4.5 for the same two data sets, Random and PDF. The cumulative distribution for both data sets is nearly indistinguishable, both for the number of chunks on the upper cumulative distribution curve, as well as the number of bytes occupied by those chunks, on the lower curve.

Note that although the divisor is 1,024 bytes, the median size of chunks is smaller, approximately 760 bytes at 50%, and total chunk storage (approximately 1,760 bytes at 50%) is larger than what is commonly expected. The expected distribution is also affected by the

lower and upper chunk size thresholds. Our analysis, explained in later sections, was limited to evaluating divisors for their efficiency and not with the goal of setting a particular expected median chunk size. However, future analysis may be of interest for system designers who are designing chunk storage based on size distributions.

Identifying Chunks An algorithm that computes a digest or a hash function over a variable-length block of data is sufficient to compute the chunk's feature, which is exactly its *content address*, a probabilistically unique identifier. Although functions such as MD5 [133], SHA-1 [113], or newer SHA standards [114] (128-, 160-bit, and larger cryptographic hash functions, respectively) would be more suitable for a production system where very low probabilities are required to help ensure global uniqueness, our prototype computes block hashes using Rabin fingerprints of sufficient degree to satisfy chunk uniqueness within the corpora that were evaluated. To evaluate the storage efficiency using a larger chunk identifier size we only need to account for the instances of identifiers that were stored and reevaluate the storage used by them.

4.4.4 Block-Based Storage

Block-based storage, or fixed-size chunks, is a degenerate case of chunk-based storage. The chunk division algorithm is trivial: it simply determines the file division at constant intervals from the start of the file. Chunk identification is unchanged. The drawback to this method is that data inserted into or deleted from a block (other than the block size itself) will prevent any data following the modification to be identified due to the misalignment of feature data.

4.4.5 Shingling

To compute a complete feature set for a file, we compute fingerprints over shingles. Shingles are overlapping substrings, W_i within a file of length s with a fixed length w. The file length is s bytes in length, resulting in $s-w+1$ shingles, $0 \leq i < s-w+1$. The window (shingle) size is fixed, for example 30 bytes. Each shingle (substring) is fingerprinted $f_{shingle}(W_i)$. Rabin fingerprinting by random polynomials [127] is commonly used for shingling [15, 46], using a fingerprint size such as 32 bits to avoid collisions across millions of objects.

Computing, collecting, and creating a subset from all features would use a large amount of temporary space, so we use min-wise independent permutations [13, 14, 46] to incrementally select features as we fingerprint shingles. This involves the use of a fixed set of k unique and randomly selected irreducible polynomials, each of which is used to fingerprint the shingle fingerprint. After each shingle fingerprint $f_{shingle}(W_i)$ is computed, k Rabin fingerprints are computed, $f_{selection_j}(f_{shingle}(W_i)), 0 \leq j < k-1$. For each j, we retain the minimum $f_{selection_j}(f_{shingle}(W_i))$ and its corresponding shingle fingerprint, $f_{shingle}(W_i)$. Upon completion, the minima are discarded. The resulting feature set is called a *sketch*. Resemblance between two sketches is simply the number of matching features divided by the total number of features.

Sketches are formed by deterministically selecting features from a large set. The goal of selection is to use randomized functions, one per element in a feature set or vector in position j, so that the selection function for each position j is independent of other randomized functions in other positions. Functions $f_{selection_j}$ are preselected once and applied throughout, and the shingles are computed deterministically. As a result, it is trivial to compute the minimum integer representation of shingle $f_{shingle}(W_i)$ from the value of $f_{selection_j}$ and then select and retain this value. We use the arbitarily chosen minimization function, but any other selection functions used consistently would provide similar behavior.

4.4.5.1 Sliding Window

The most efficient shingling parameters are a tradeoff, and depend on the type of data. We experimented with window size, w, from 8 bytes to 256 bytes, and determined that the overall compressed size from the *delta compression between similar files* algorithm (DCSF) varied slightly, but for each data set there was a minimum. Our experiment used ten versions of the Linux kernel source code, versions 2.4.0-2.4.9, 88,322 files, 1.00 GB in size. Figure 4.6 displays the number of files matched by DCSF for each number of matching features. We set a low matching threshold to 1 of 16 fingerprints, with a sharp peak at 2 of 16. The even larger peak at 16 of 16 matching features is due to the highly similar data set of related text source code files, at 75,524 matching files ($w = 8$) down to 72,567 ($w = 256$). This "bathtub curve" indicates that many files exhibited high resemblance and also that many other files were compressed that had little, but non-zero, resemblance.

As one would expect, small window sizes (*e.g.* $w = 8$) allowed DCSF to match files across the spectrum of resemblance, so the low-resemblance peak was lower and mid-resemblance was higher. Conversely, large window sizes decreased matches in the mid-range (4–14 of 16 features matching).

While the difference in granularity (and coverage) varies, there are two other effects to be noticed. First, the cumulative distribution of files matching above all possible resemblance values in Figure 4.7 indicate the gradual nature of matching ability of fine grained windows ($w = 8$). The second, and more important result, is that the total storage, namely the rightmost points for all values of w, is nearly identical. For highly-resembling data, the window size does not have significant impact on compression rate.

Figure 4.8 illustrates this point as well. The edge on the upper right represents the total amount of data stored across all resemblance. The slight inflection reflects favoring window

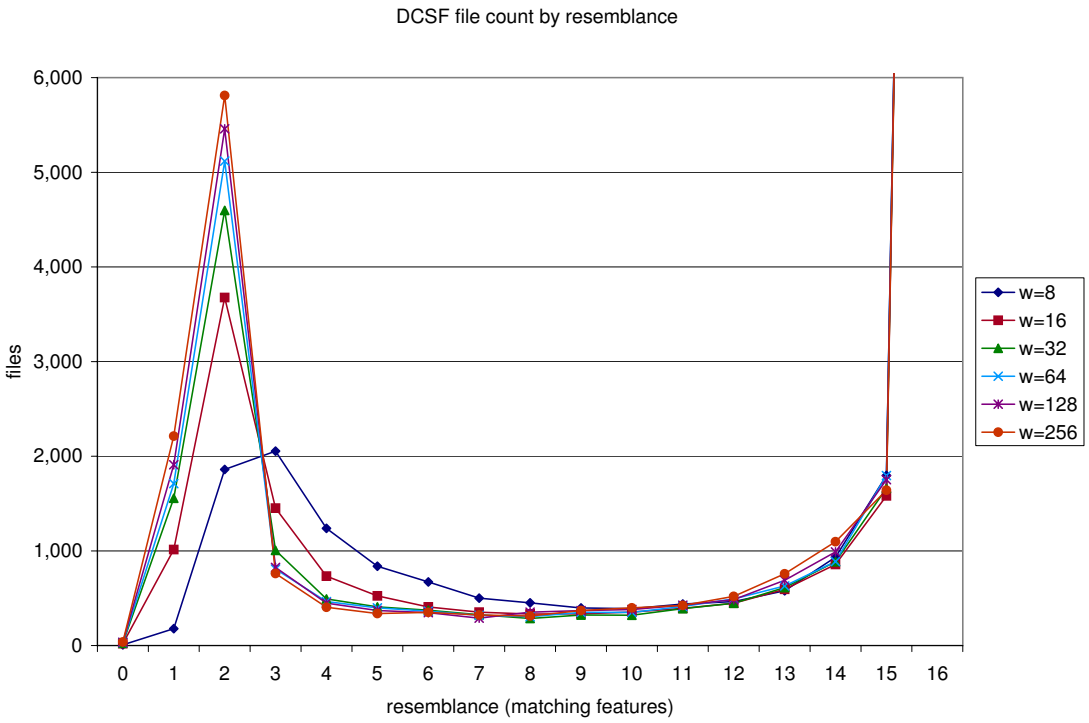


Figure 4.6: DCSF filecount by resemblance and window size of highly resembling data, Linux 2.4.0–2.4.9 kernel source; (72,567 to 75,524 files at 16 of 16 matching features)

window size (bytes)	$w = 8$	$w = 16$	$w = 32$	$w = 64$	$w = 128$	$w = 256$
bytes (1000s)	47,729	47,298	47,015	46,864	47,016	47,575
% over best	1.845%	0.925%	0.321%	0.000%	0.324%	1.517%
% of uncompressed	4.426%	4.386%	4.360%	4.346%	4.360%	4.412%

Table 4.3: DCSF compression based on window size (1,078,315,981 bytes input)

sizes around $w = 64$. Table 4.3 and Figure 4.9 list the the compressed sizes (less than 4.5% of the original input), and the variation across the window sizes is less than 1.85%.

Other data sets also exhibit similar variations, with optimal compression ranging in the range of 16–128 bytes. As is the case with selecting chunk size divisors, no single parameter works best across all data sources. More importantly, a single window size must be selected for

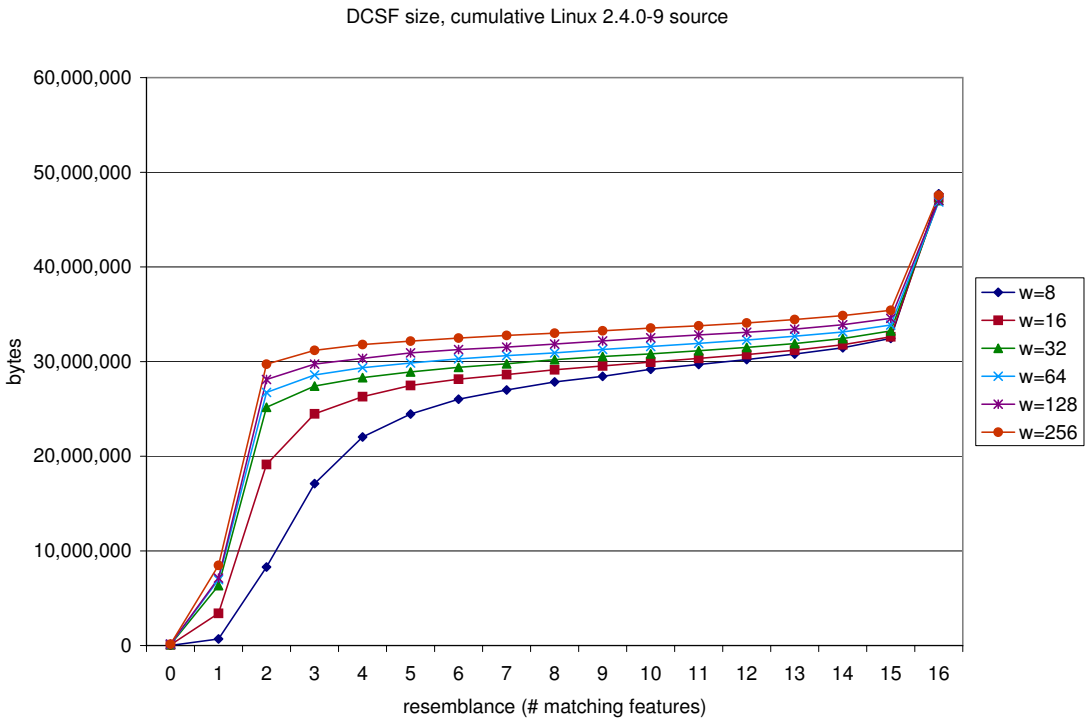


Figure 4.7: DCSF cumulative file count by resemblance and window size

all data in order to compute features deterministically; fortunately, for window sizes within this range the data compression rates do not vary significantly.

4.4.6 Superfingerprints

One refinement to further summarize the sketch is to compute superfingerprints, or fingerprints of features. Superfingerprints are fingerprints of a fixed number of features [15]. For example, let the string SF_0 be the concatenation of $f_{shingle}(W_i), 0 \leq i < l$. Then $f_{super_0}(SF_0)$ is the first superfingerprint, $f_{super_1}(SF_1)$ the second, and so on. Figure 4.10 shows the how superfingerprint $S_{0..3}$ is simply a fingerprint with input string of concatenated fingerprints $f_{shingle}(W_i), 0 \leq i < 4$. This reduces the sketch size by a factor of l. Because the features are independent, a sin-

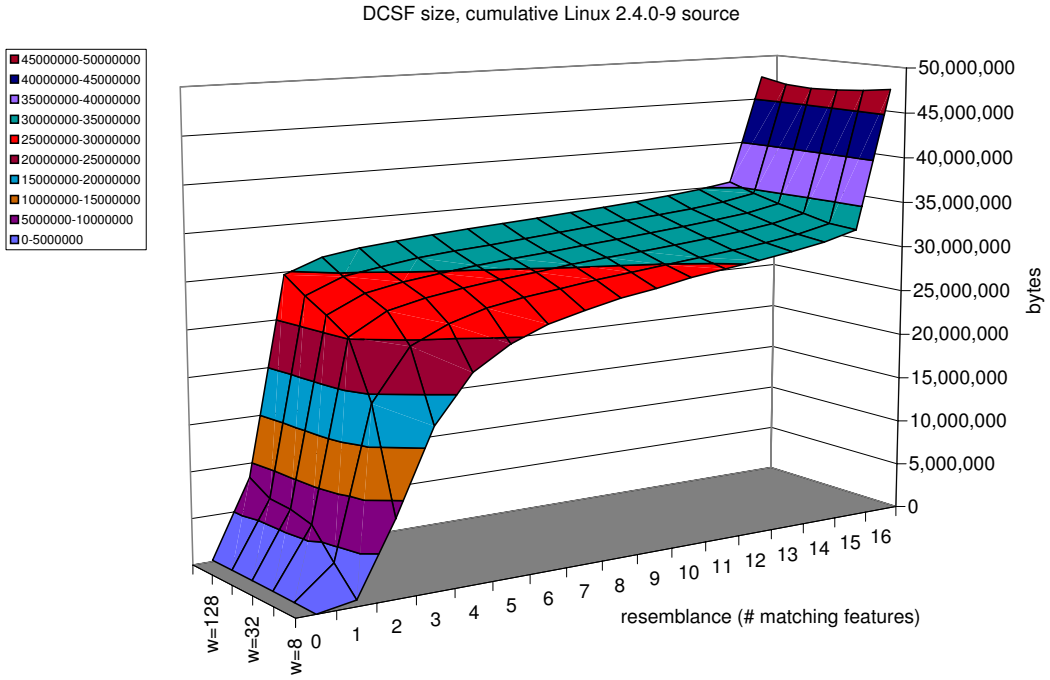


Figure 4.8: DCSF cumulative file count by resemblance and window size (3D)

gle superfingerprint that matches a corresponding superfingerprint in another sketch reduces the probability of locating a sketch with low resemblance while still being able to detect sketches with high resemblance [46]. In Chapter 5 we elaborate on the use of superfingerprints in the similarity detection phase of PRESIDIO.

4.4.7 Evaluation

We measured the hashing performance of selected digest and fingerprinting functions. Table 4.4 lists the functions, the size of the fingerprint, and the computational throughput. (File sizes were 100 MB and 256 MB, measured with 10 microsecond resolution, pre-cached file data.)

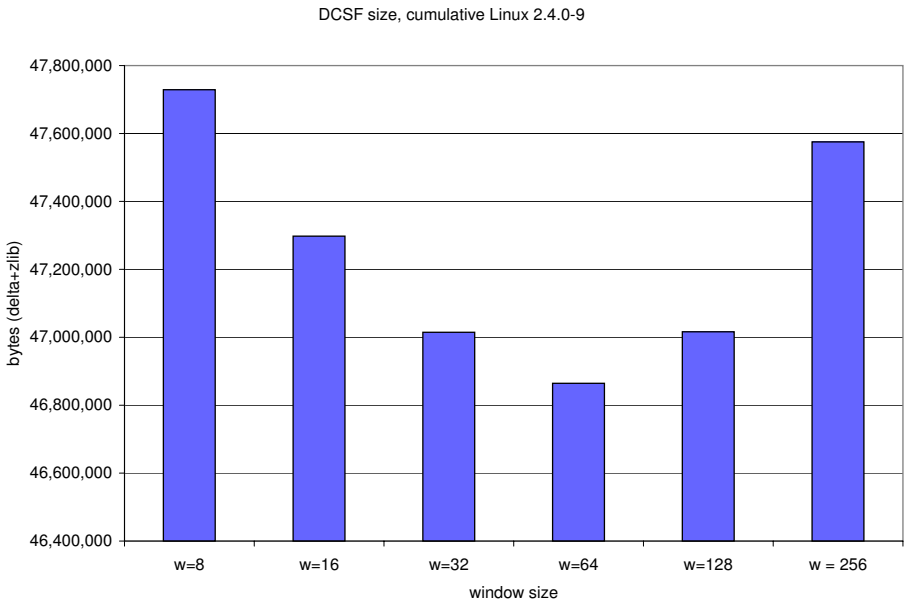


Figure 4.9: DCSF storage efficiency based on window size

The program versions are the following:

MD5 : GNU *md5sum* (coreutils) 4.5.3

SHA-1 : GNU *sha1sum* (coreutils) 4.5.3

Rabin fingerprint : UCSC Deep Store Rabin fingerprinting library, 12/2004

chc32 : Chunk Compression program, 32-bit version

shingle32 : Shingle and superfingerprinting program, 20 features

cat : The Unix *cat* program, writing to the null device

The Rabin fingerprinting program operates by appending a single byte or a (32-bit) word. The efficient execution of this function is due to precomputed table lookup; implementa-

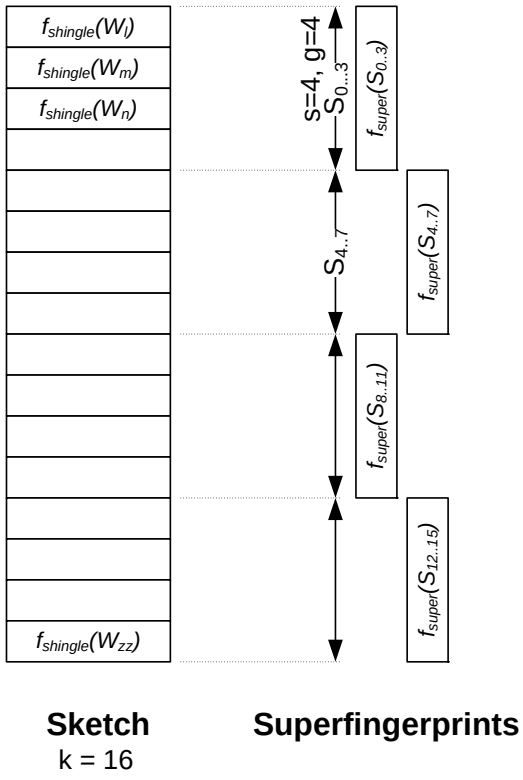


Figure 4.10: Sketch and superfingerprints

tion details can be found in Appendix A. For each program, features were computed from sample input files that were fully cached by the operating system buffer cache. Whole-file hashing using MD5 and SHA-1 were much faster than disk bandwidth, suggesting that I/O bandwidth would be the limiting factor in implementation. Likewise, Rabin fingerprinting performance computing a single fingerprint over the entire data file is significantly faster. The "by bytes" operation appends data by bytes to the fingerprinted string, and "by 32b words" appends data by 32-bit words. The chc32 program computed both strong and weak hashes (one for the chunk fingerprint and one for the sliding window to determine division point) and also stored chunk data for its output, written to the null device. The shingle32 program computed a feature vector

	hash size (bits/bytes)	MB/s
MD5	128b/16B	174.8
SHA-1	160b/20B	79.5
Rabin fingerprint (by bytes)	32b/4B	272.5
Rabin fingerprint (by 32b words)	32b/4B	1,528.0
Rabin fingerprint (by bytes)	64b/8B	75.2
Rabin fingerprint (by 64b words)	64b/8B	1,514.6
chc32	32b/4B	36.4
shingle32 (before optimization)	32b/4B	0.5
shingle32 (after optimization)	32b/4B	19.7
cat > /dev/null	N/A	45.0

Table 4.4: Feature selection performance for selected digest and fingerprinting functions using ProLiant (PL) hardware

of 20 bytes.

Programs computing shingle feature sets using the min-wise permutation algorithms are computationally expensive. Our initial implementation reached 0.46 MB/second (listed as 0.5 MB/second in Table 4.4), but we were able to improve the throughput dramatically by more than 42 times through careful programming. Such improvements are necessary if resemblance detection is to be practical in an environment that evaluates different compression algorithms. We also note the performance of reading a file with the *cat* program on an uncached file, at 45.0 MB/second.

Our test hardware machines, listed in Table 4.5 include CPQ and PL. We used the single-node HP ProLiant (PL) server for the experiments in Table 4.4. As is evident by the high throughput of our implementation of Rabin fingerprinting, non-cryptographic functions can have a significant performance improvement over cryptographic hash functions.

Description	CPQ	PL
Hostname	cerium	deep-store-8
CPU	Compaq Evo N620c	HP ProLiant DL320 G2
Platform	Cygwin/XP	Red Hat 9 Linux
Processor	Pentium M	Pentium 4
Clock Speed	1.6 GHz	2.66 GHz
L2 Cache	1,024 KB	512 KB
Frontside cache bus speed	400 MHz	533 MHz
RAM	1,024 MB	2,048 MB
Memory speed (PC)	PC 2100	PC 2100 ECC
Hard disk drive	Toshiba MK6026GAX (ATA)	Seagate ST380023A (ATA)
size, speed, cache	60 GB, 5400 RPM, 16 MB	80 GB, 7200 RPM, 2 MB
Disk file system	NTFS	NFS/GPFS/ext3

Table 4.5: Hardware

4.5 Discussion

Data identification through fingerprinting and feature selection is the foundation to the solution of our archival storage problem. A practical implementation is necessary for adoption in environments where data storage bandwidth is a primary concern. The algorithmic improvements to detect similar data are dependent on probabilistically unique identifiers. And the tradeoffs between data identifier size and retrieval accuracy are important due to the incumbent overhead of storing metadata information.

A conflict exists between choosing parameters and algorithms for "optimal" behavior and the wide variety of input data provided to a storage system. While a range of fingerprinting parameters might work for one data set, it might not work for others; however, it is desirable and even necessary to fix parameters. Our experiments have shown that the range of parameters is somewhat flexible. For example, the sliding window size of shingled binary data plays a much smaller factor than the compression method that is employed. Because compression behavior is data dependent, there is a stronger need to select the best compression method, and a need to

use multiple methods within the same system.

During our investigations, we discovered that data fingerprinting methods were extremely useful and could be applied in a number of ways; the performance would make a tremendous difference between a practical solution and a non-starter. For example, the "inner loop" for computing a Rabin fingerprint was a bottleneck, as was fingerprinting to perform min-wise selection from a sketch. Careful programming improved performance by more than an order of magnitude.

4.6 Summary

In this chapter we have described algorithms that we use for computing and selecting features from binary storage objects, including whole file hashing, variable-sized chunking, data fingerprinting over shingles, and superfingerprints over blocks of data. Because our solution examines features over a range of object sizes, we have examined feature performance and the coverage of fingerprinting methods. Fingerprinting sliding windows does not have a single optimal window size; we have evaluated different sizes to show that for some data types different sizes affect overall storage efficiency; however, there is no one size that provides an optimal storage efficiency over all data.

We have created a chunk compression program called *chc* that divides files into chunks. Chunk size distribution can be parameterized by different sizes, including minimum, maximum, and divisors. The distribution for highly structured file types is similar to that for random data, but some variation occurs for commonly identified blocks. To prove the capability of both chunk and shingle-based feature selection, we have implemented fingerprinting algorithms using shingles and superfingerprints that are capable of yielding throughput on par with that of disk devices that are available today.

Chapter 5

Finding Data

In this chapter we focus on finding identical or similar data within the archival store. We show we use different *similarity detection* algorithms over input file data. The output from the similarity detection algorithms are then used in the next stage in PRESIDIO to compress data by eliminating redundancy. Data compression over large scale storage trades off the cost of detecting similar or identical data at fine or coarse granularity against the effectiveness of the the detection methods and data compression algorithms.

5.1 Overview

Efficient archival storage compression methods use similar phases in processing. The different *efficient storage methods*, or ESMs, we use within our framework eliminate redundancy using techniques like suppressing storage of common data or by storing delta encoded files. We identify two common compression phases which we can use within the framework, namely *similarity detection* and *redundancy elimination*. These follow *feature selection*, in which fingerprints are computed, but before *recording to disk*, where compressed data are writ-

ten to the storage medium.

The qualitative differences in similarity detection algorithms most strongly distinguish the differences in efficient storage methods. We recognize this common phase makes the tradeoff between highly accurate detection of identical data at little space and execution cost—to a middle ground of heuristic algorithms which find highly similar data with some accuracy and low cost—to complete comparisons of pairs across the entire archive at high cost but diminishing returns. Exploiting the best aspects of each ESM is a key part of our solution.

This chapter is organized in the following manner. First, we provide an overview of similarity detection. Next, we examine methods to detect similar and identical data using hashing and resemblance detection mechanisms in more detail. Finally, we close with a summary of similarity detection within the context of our solution.

5.2 Similarity Detection

We define *similarity detection* as the method by which two or more similar or identical *data objects* are located from within a large corpus. In practice, we use data objects that are byte-aligned, contiguous binary string data, such as a partial file or a whole file. The data objects are identified or named by their content address. Thus as a prerequisite to using content addresses, we consider the content immutable.

The objects stored in a content-addressable store have both *virtual* and *real* representations, which are uncompressed and compressed, respectively. A client of the storage system relies on the virtual file image. The redundancy elimination mechanism uses both virtual and real file representations, and the real object data size is usually smaller than whole files.

Similarity detection is a key phase in data compression of large-scale data stores because it is what makes the elimination of redundant data possible. We compare and contrast

PRESIDIO data compression to stream compression, commonly used to compress individual files.

Stream compression algorithms work by first identifying similar data. For example, sliding window dictionary based compressors like Lempel-Ziv [177] continuously update a string dictionary. As new uncompressed data is read in, the compressor looks for previously encoded strings that match incoming data. On a much larger scale, our PRESIDIO detects similarity using multiple methods including hashing and feature selection algorithms, and then a separate phase eliminates redundancy by encoding data before it is written to disk.

Stream compression algorithms and PRESIDIO operate over different ranges of data size. Whereas a stream compressor examines substrings in short windows and retains string dictionaries that fit in real memory—for example kilobytes or small numbers of megabytes—PRESIDIO extracts features from files of arbitrary size. Stream compression does little or nothing to eliminate redundancy across distinct data objects. In contrast, PRESIDIO identifies similar uncompressed data across its entire corpus. Both fine-grained stream compression and large-grained similarity detection are complementary. PRESIDIO incorporates both types of compression into its architecture.

The number of methods that can be developed for detecting similarity is unlimited. The effectiveness of a single method varies with the type of input data. Because our goal was to develop a general-purpose storage system, we employed methods that compress a variety of data types. Content-specific methods exist, such as those which identify similarity between text-based web pages [15, 39], but their shortcoming for our storage problem is that they are not designed for non-textual data.

Similarity detection algorithms evaluate features such as fingerprints, superfingerprints, and sketches, which are selected from content in a previous phase. We use features

because their small size enables high performance of similarity detection at low cost. If we did not, then it would be impractical to compare files. For example, in a corpus of one billion (10^9) files, each with average size of 5,000 bytes (conservatively, twice the median file size in a recent survey [154]), comparing each new file against all existing files would take $n(n+1)/2$ comparisons, and each comparison at best would read at least $5,000 \times 10^9 = 5 \times 10^{12}$ bytes ≈ 5 terabytes. We will show how we reduce both the number of comparisons as well as the cost for each comparison by using features instead.

The detection algorithms use metrics such as the *resemblance* between two files and compression efficiency instead of theoretical differences like the *(Levenshtein) edit distance*.

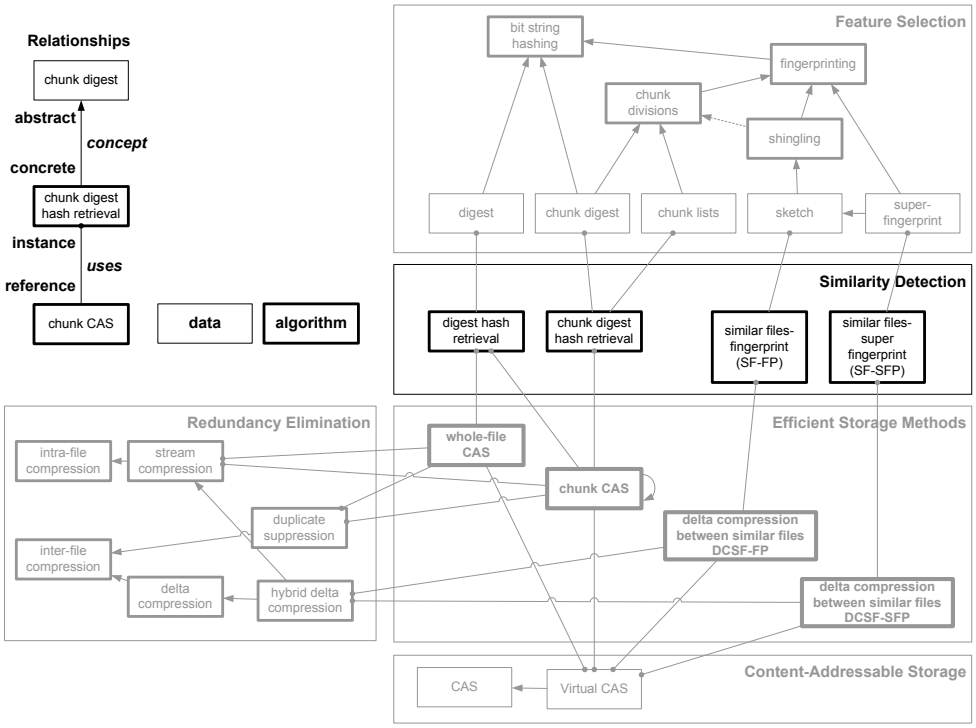


Figure 5.1: Similarity detection algorithms

Figure 5.1 illustrates the different similarity detection methods and their relationships to dependent feature selection processes. The main methods we describe include *digest hash retrieval* or *whole-file hashing*; *chunk digest hash retrieval*, or *chunking*, *SF-FP* or *similar files with fingerprinting*; and *SF-SFP* or *similar files with superfingerprints*. Each method exhibits different similarity detection properties.

5.2.1 Parameters used in similarity detection

Each similarity detection method we incorporate uses a set of different parameters to maintain low storage and computational overhead.

virtual file an original file presented by a client.

virtual CAS object a virtual object stored in a content-addressable store.

real CAS object a real instance of data stored within virtual content-addressable store.

feature data the data from which a feature is computed. coverage

feature coverage size the size of data from which a single feature is computed.

coverage the fraction of data from which features are computed.

feature set (or feature vector) the set of features computed from the content of a virtual CAS object.

feature set size the cardinality of a feature set.

fingerprint size the size of a fingerprint, usually represented in bits.

The virtual file is the original file presented to the CAS for storage. Because the virtual *file size* can vary greatly in a data set, our storage system may subdivide it into a sequence of virtual CAS objects in order to further detect similarity among blocks.

The virtual CAS object may be a whole file, or it can be smaller, like a chunk or delta file. The virtual size measures the amount of the original instance of the file.

Real CAS objects always provide backing for the virtual CAS file image. The real object size may be the same as the virtual size. The incremental real object storage cost may be zero, in the case when file storage is suppressed because an exact copy of a file has already been stored.

Features are computed from *feature data*, *e.g.* a whole file, variable sized chunk, fixed-size chunk, sliding window.

Coverage varies from 100%, in the case of a whole-file digest, to a smaller fraction in the case of a fingerprint sketch. A whole-file digest accounts for every byte in its file. In contrast, a sketch may only cover $k \times$ *feature coverage size* $\div$ *file size*. Although the coverage for a sketch used to detect similarity between two files may be low, it does not significantly alter the probability that deterministic feature selection algorithms will select the same features.

The feature set size is usually denoted in our work by k, the number of chunks from a file subdivided into chunks or the number of fingerprints computed and stored in a shingle-based sketch.

The size of fingerprints is established once during the design of the system and fixed, for example, at 32 bits.

5.2.2 Comparing similarity detection properties

To illustrate the way similarity detection is used, in Table 5.1, we list some of the similarity detection properties that are characteristic of each efficient storage method and compare qualitative differences.

	ESM	feature	coverage	fp	size	k	selection
1	Whole File	file	whole file	MD5	16	1	hash file
2	Chunk	chunk	128–8192 var.	RF32	4	m	hash chunk
3	Chunk	chunk	128–8192 var.	RF64	8	m	hash chunk
4	Chunk	chunk	128–8192 var.	MD5	16	m	hash chunk
5	Chunk	chunk	128–8192 var.	SHA-1	20	m	hash chunk
6	Chunk	chunk	4096 fix.	SHA-1	20	K	hash block
7	DCSF-SFP	s.w.	8–32 fix.	RF32	4	$\{2^n,2^m\}$	H-SFP
8	DCSF-SFP	s.w.	8–32 fix.	RF32	4	$\{84,6\}$	SFP
9	DCSF-FP	s.w.	8–32 fix.	RF32	4	K	MWP
10	DCSF-FP	s.w.	8–32 ch.	RF32	4	K	MWP

ESM	efficient storage method
feature	data (content) from which a feature is computed file entire file chunk variable- or fixed-sized chunk s.w. Rabin fingerprint sliding window
coverage	number of bytes covered by a single feature; may be variable-sized (var.) or fixed-size (fix.)
fp	fingerprint algorithm; digest or Rabin fingerprint RF32 (32-bit) or RF64 (64-bit)
size	size of a feature in bytes; equal to fingerprint size
k	feature set size; in superfingerprint, $\{k, l\}$ represent the cardinality of the initial sketch size k, followed by the cardinality of supersketch size l; (in Section 4.4.6 group size $g = k/l$)
selection	feature selection process H-SFP *harmonic superfingerprint* SFP *superfingerprint* MWP *min-wise independent permutations*

Table 5.1: Comparing efficient storage methods and their similarity detection properties

Each efficient storage method (ESM) uses one similarity detection method. Since PRESIDIO is a hybrid system, it aims to use the best properties of each method. We briefly discuss the advantages and shortcomings of each method's ability to detect similarity. The methods in Table 5.1 are in approximate order of coverage size, starting with (1) whole file hashing, and ending with (10), delta compression between similar files with fingerprinting.

5.2.3 Identical Data

Detecting identical data using probabilistically unique hash functions is straightforward: an input file is interpreted as a long binary string and the digest function computes a single hash value over this input.

Whole file hashing (Table 5.1, line 1) computes a feature from a single coverage area, the whole file. The advantage is that the hash value, using strong hashing, is easy to compute. Its main shortcoming is that the probability of detecting an identical file is high (with very low probability of a false collision), but its probability of detecting a similar file is zero.

The *chunk* ESM (Table 5.1, lines 2,3,4,5) is similar to whole file hashing, but first subdivides the files into pieces. This has multiple advantages over whole file hashing. The first is that by subdividing a file into smaller pieces, in the event of small differences between two files, the probability of detecting identical chunks is higher than the probability of only detecting identical files. A second benefit is that multiple chunks might be detected as identical within the scope of a single file. The main drawbacks to this method are that the number of chunks incur overhead, the methods for subdividing chunks deterministically are chosen arbitrarily but must be used consistently across all chunk storage. Methods 2-5 differ only in the size of the (strong) hash method.

A degenerate case of the *chunk* ESM occurs when chunks are of fixed size (Table 5.1,

line 6). In this case, fixed size blocks are selected, typically dividing files at constant intervals starting from the first byte of the file. This is the method used by the Venti file system [125]. The difference between fixed-size and variable-size chunks is that the subdivision method is trivially specified, and that use of the underlying storage system may be more efficient when the same fixed size is used. The drawback is that insertions or deletions that are small (or more precisely, not of exactly the size of the fixed size block) in a file prevent similarity detection of changes subsequent to the position in the file. Because this block variation does occur in practice [124], the chunk-based method is efficient when per-chunk overhead is less than the space that is saved over fixed-sized blocks. Our CAS is implemented using large flat files, and stands to gain little storage efficiency by using fixed-sized block storage.

5.2.4 Similar Data

Superfingerprints are fingerprints over features stored in a sketch of size k. Previous work that detects web document similarity [15] used superfingerprints with a fixed group size by fingerprinting s groups of g features. Other examples have permuted the k features to produce *megashingles* or *super-superfingerprints*. In all cases, the group sizes, g, were fixed.

The DCSF-SFP method (Table 5.1, lines 7,8) depends on the generation of sketch sizes of cardinality k. Features are Rabin fingerprints, computed over data in sliding windows. The sliding window itself is chosen once and fixed for all time, from empirical data. Two examples are shown, the first being *harmonic fingerprinting*, described below. In the example of line 8, by selecting initial sketch size $k = 2^n$, and superfingerprint sketch size $l = 2^m$, $m < n$, then searching for matching sketches is in a smaller space. The degenerate case, $m = 0$, or superfingerprint sketch $k = 1$, produces a single superfingerprint which can detect highly similar files with high probability. We evaluate harmonic superfingerprints in further detail below.

The parameters in Table 5.1, line 8, $\{84,6\}$, (sketch size and number of superfingerprints, respectively) were used to detect resemblance between web pages [15, 99] is provided for comparison, but due to the large number of fingerprints (84) and subsequent storage overhead in the initial sketch, we do not consider these parameters for our use because space efficiency is of primary concern.

The DCSF-FP methods (Table 5.1, lines 9,10) are prerequisite computations to DCSF-SFP (7,8), but in practice we compute both at the same time. In the case of DCSF-FP, searching for fingerprints is an indexing problem. Searching multidimensional spaces may take considerable time or space.

5.2.5 Harmonic Superfingerprints

We further refine superfingerprints to progressively detect similar files with lower resemblance using *harmonic superfingerprints*. For identical, and highly similar documents, a small number of superfingerprints, s, is desirable because it reduces the search space. Earlier work have matched one, two, or more superfingerprints to *increase* the specificity of detection, but some data sets can be compressed significantly using small numbers of superfingerprints covering an entire sketch. The result is that we use the superfingerprint covering the entire sketch in lieu of a whole-file digest to perform a hash lookup, and other superfingerprints to search in low-dimensional spaces.

First we precompute harmonic superfingerprints at the time we compute a sketch, then we progressively search the superfingerprint space to find sketches of decreasing resemblance. Let $s = 1$ and $g = k$, in other words, the superfingerprint is computed over all features in the sketch. Next, compute $s = 2$, $g = k/2$, so that each superfingerprint is computed is computed over half the sketch and continue in this manner, doubling s each time. (For convenience we

select $s = 2$, but it can be non-integral and scale arbitrarily.) Figure 5.2 illustrates the sketch ($k = 16$), ranges of the superfingerprints and the resulting superfingerprint set.

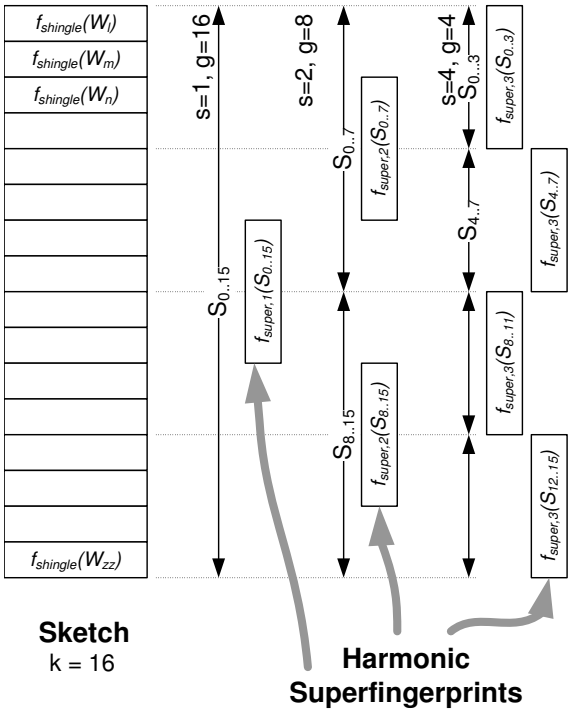


Figure 5.2: Sketch and harmonic superfingerprints

We can compute the probability that a file matches another file with resemblance at least p. First we divide the sketch of k features into s groups of g. The probability of an individual feature matching the corresponding feature in another sketch is p, the probability of a superfingerprint over g (independently selected) features is p^g. We have s groups, so the probability of one or more groups matching is $1 - (1 - p^g)^s$.

Each curve in Figure 5.3 shows the probability that a pair of files with a given resemblance (horizontal axis), is detected by a *single* superfingerprint out of s. The sketch size overhead is fixed at k features (in this example, $k = 32$), resulting in k/s features per group. If a single fingerprint is computed over all features in the sketch (rightmost curve), then 100% of

the files with high resemblance are detected while the leftmost curve; it also filters out pairs with less than 90% resemblance, detecting less than 4% of them. (Note that the leftmost curve, $g = 1$, has $k = 32$ features, the same as a feature set. The difference is that a superfingerprint set will contain one fingerprint of a fingerprint. Comparing two sets of superfingerprints still requires $s = 32$ tests requiring $s = 32$ comparisons between two sketches and a high probability of detecting low as well as highly resembling files.)

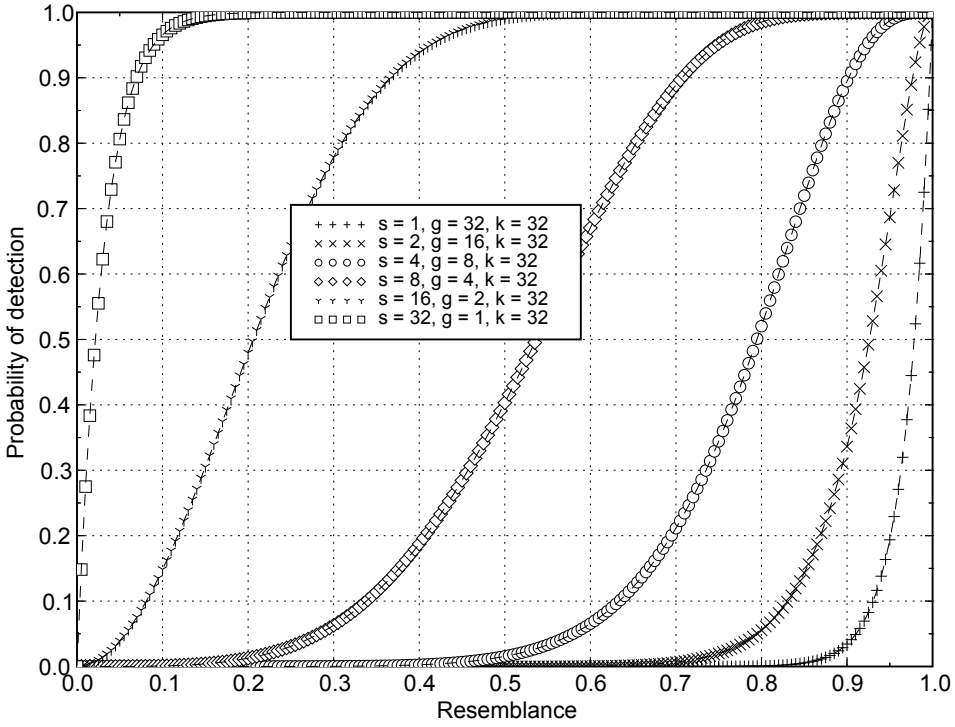


Figure 5.3: Probability of detection with one superfingerprint, $k = 32$

For comparison, we evaluate the probabilities a single superfingerprint will match files above a resemblance for a larger sketch size. When the sketch size is increased to 128 features ($k = 128$), the probability of detection takes on a different shape for the group sizes.

With large group sizes ($g = 128$), the probability of finding highly similar files is very high and it also restricts matching files below 95% resemblance.

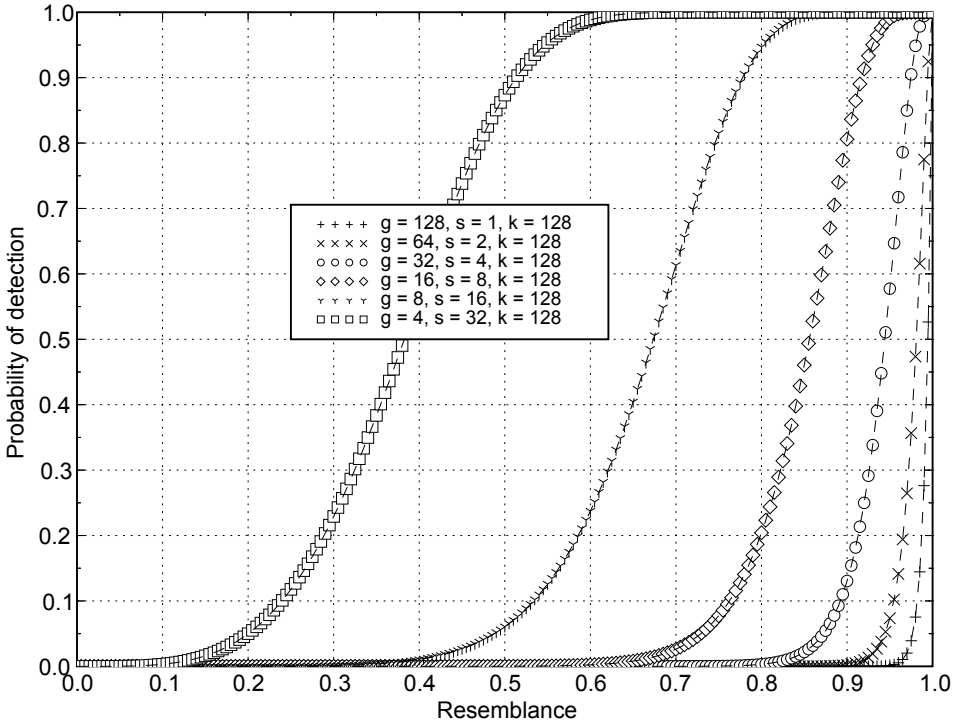


Figure 5.4: Probability of detection with one superfingerprint, $k = 128$

The computational cost to compute a superfingerprint is linear, $O(k)$, in the size of the sketch. Compare this with the complexity to compute a sketch using min-wise independent permutation feature selection $O(sk)$, where s is approximately the size of the file. Thus for non-trivial files, the computational cost for superfingerprints is negligible.

Searching is straightforward: we first search with the $s = 1$ superfingerprint. Although this is similar to whole-file hashing, it is tolerant of slight variations in the file. Conversely, identical files will produce identical superfingerprints. Next, we search the superfingerprints in

the dimensional space $s = 2$ and so on until we find a sketch. Larger than these trivial cases, the problem returns to a problem similar to an n-dimensional keyword retrieval. Traditional methods using an inverted keyword index [170] requires large amounts of memory.

We use harmonic superfingerprints to progressively search the space by finding highly similar files quickly and moderately similar files with more time. The time to find highly similar files in our example is $1/k$, which is significant. If the first harmonic superfingerprint fails, i iterations with $s = 2^{i-1}$ results in $2^i - 1$ superfingerprint lookups. Increasing i by more than one can reduce the total number of tests, for instance if $s = [1,4]$, in the best case we speed up by 32 and in the worst case by 5.

If no sketch is found, we fall back to slower methods, such as comparing sketches.

5.2.6 File Resemblance

The wide diversity of archival data necessitates different storage compression methods to achieve a high level of space efficiency. In this chapter we first examine data compression as it applies to archival storage, describe how different types of data and types of compression make it difficult to use a single method for all data, and then evaluate the methods to contrast the differences. Finally, we discuss how we can exploit these differences by integrating the methods into a system that seeks to capture the best of each storage method.

5.3 Evaluation

5.3.1 Whole-file hashing

In practice, whole-file content-addressable storage is useful when it is likely that identical copies of the file will occur. A system such as EMC Centera [42] will detect such files.

The system performs a hash function such as MD5 or SHA-1 on the entire contents of the file, and then checks to see if an identical hash already exists. If it does, then recording the file to the storage device is suppressed, otherwise a new copy of the file is stored.

The resulting savings in storage is directly related to the probability of multiple identical copies of files. If they are common, then compression will be good, if they are rare then little savings will be realized.

5.3.2 Chunking

When there is not an identical copy of the file, other methods can be employed, such as chunk-based content-addressable storage (*chunking*). Whole-file CAS only finds exact copies. Chunking divides files deterministically into variable-sized chunks of which single instances are stored. The technique first used in the low-bandwidth file system (LBFS) [110] divides files deterministically into chunks and then identifies chunks by hashing their contents. A file is stored in two parts: a list of the chunk identifiers and the chunk data. Chunks are stored as single instances of content-addressable objects. The fine-grained single-instance chunk storage improves on the efficiency of whole-file CAS when files are similar but not identical.

Compromises in chunking implementation are its shortcoming. To ensure identical chunks can be found when new files are added to a system, chunking behavior must be deterministic, which in turn requires that parameters be selected once and for all data. Unfortunately, a fixed set of parameters does not optimally compress all data. Chunking metadata (chunk lists) increases storage overhead which makes it less effective than whole-file CAS when unique files are stored.

5.3.3 Superfingerprint

The superfingerprint compression rate for a single group of size $s = k$ is the same as the compression for files compressed for resemblance $r = 1.0$ (excepting hash collisions). A simple example using Linux source code versions 2.4.0–2.4.9 resulted in 84% of the files matching exactly by superfingerprint.

5.4 Discussion

Our original intention for "finding data" was to find a solution to more general classes of problems: how to store metadata information in a traditional file system to facilitate search; how to identify clusters of data and then find similar clusters; how to index and retrieve similar data using traditional information retrieval and web search techniques. These problems present unique challenges and extensive areas of research have branched into peer-to-peer networks, indexing the entire web, distributing storage capabilities across machines in widely distributed areas, improving content-aware search in personal file systems, and so on.

We proposed using shingling and data clustering methods; the clustering algorithms are numerous [75] and were used in a search engine, but in a system where storage frugality and incremental storage are primary goals, the clustering methods available today are not well matched for our problem statement. Although we first envisioned clustered, or "associative" storage to be the solution, we have turned toward heuristic engineering solutions where approximate metrics and low-cost and high-return will be most practical. In fact, harmonic superfingerprinting only developed as an observation that a generalized indexed retrieval method would be costly and that most of the benefit in data compression will come from highly similar data.

An apparent disparity between chunking and delta compression forced us to pose the question which direction we should take. On one hand, chunking was attractive due to its simple data identification model and chunk retrieval model. Chunk hierarchies would offer benefit for similar data by sharing chunk lists or trees. On the other hand, we had previous experience that delta compression could express fine-grained differences very efficiently. Our comparison between the two was intended to discharge one approach. Unfortunately, experiments were inconclusive, leading us to embrace not just both these methods but any others into a common storage framework. But before doing so, we would need ways to find similar data regardless of the data compression method being used. The result of our design work resulted in the need to identify the common, but implementation-dependent "finding data" step described in this chapter.

Chapter 6

Compressing Data

In this chapter we focus on compression through *redundancy elimination*. Data compression used in a large scale storage is dependent on identifying and finding data, and then later records data into a low-level content-addressable store. Redundant data is identified as identical or similar, using the methods described in previous chapters. Redundant data is eliminated by encoding original data to occupy less storage than the source input or by suppressing storage completely.

6.1 Overview

The PRESIDIO redundancy elimination phase, which uses a multitude of *efficient storage methods* (ESMs), is a practical application of data compression algorithms. Our contribution to the storage area is to use existing data compression algorithms in a manner that produces compression benefits in a large-scale system. The main compression algorithms we apply on whole or partial files are *suppression of storage*, *stream compression*, and *delta compression*, or a combination of those methods.

Whole-file hashing and chunk-based storage suppress storage of multiple instances of identical data. Delta compression stores *reference files* and *delta files* and relies on file reconstruction to produce the original input. Although single-instance storage and delta storage are space-efficient, they introduce new data dependencies not present in stream compressed data alone. In turn, the data dependencies affect the reliability model. As a result, new data protection designs will be needed to provide long-term storage reliability.

This chapter is organized in the following manner. First, we provide an overview of data compression by eliminating redundancy in identical and similar data. Second, we describe ways to eliminate redundancy within and across stored objects. Third, we evaluate compression methods for their storage efficiency. Finally, we close with a summary of redundancy elimination within the context of our solution.

6.2 Redundancy Elimination

The second half of the data compression process is made up of the *coder*, a step that creates a more compact encoding from the uncompressed data and then recorded. The two methods we use to reduce or eliminate redundancy are through *intra-file compression* and *inter-file compression*. Figure 6.1 illustrates the different compression methods used within the redundancy elimination phase.

6.2.1 Intra-file Compression

Intra-file compression consists primarily of stream compression, as described in Section 2.5.1. Data compression within a single file is a mature research area. Our system assumes the availability of stream compression and decompression programs with suitable properties for the content we wish to store.

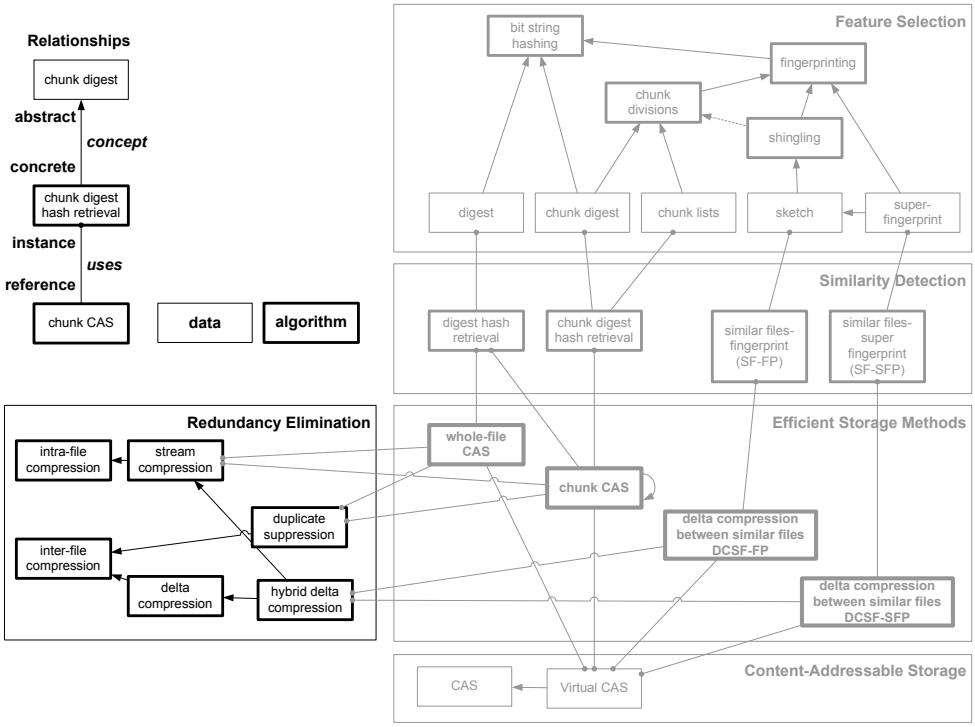


Figure 6.1: Redundancy elimination methods

When used together, inter-file compression may affect intra-file compression. For example, when using *chunking* algorithms, the size of contiguous uncompressed data in a chunk to be compressed is smaller than a file. Because stream compressors work to amortize the cost of collecting and storing internal dictionary data across an entire file, when a file is subdivided, the amortizing efficiency is lost. Hence, the sum of compressed chunk sizes may be larger than the compressed size of the file comprised of those same chunks.

In other instances, such as suppressed storage of identical data or delta-compressed data, intra-file compression is independent of inter-file compression.

6.2.2 Suppression of Multiple Instances

Data that is identified by content stores a single instance of data. When subsequent operations to store data with the identical content, the same content address is used. The process of *suppressing storage* and referencing the single instance multiple times compresses by eliminating duplicates.

6.2.2.1 Whole-File Hashing

Whole-file content-addressable storage is useful when it is likely that identical copies of the file will occur. A system such as EMC Centera [42] will detect such files. The system performs a hash function such as SHA-1 on the entire contents of the file, and then checks to see if an identical hash already exists. If it does, then recording the file to the storage device is suppressed, otherwise a new copy of the file is stored.

The resulting savings in storage is directly related to the probability of multiple identical copies of files. If they are common, then compression will be good, if they are rare then little savings will be realized.

Metrics The storage efficiency, u, is simply

$$u = \frac{\text{real storage size}}{\text{virtual storage size}}.$$

In the case of whole-file hashing, this efficiency is determined by a simple resemblance, r between two files. It is binary: either $r = 1$ or $r = 0$. For $r = 1$, this metric is a probabilistic estimate: the probability of a strong hashing function collision of content addresses of size k bits is less than approximately $2^{-k/2}$.

Thus if f is the size of an input file, the storage size is rf; either it is full size of the file ($r = 1$) when the file is first stored, or zero ($r = 0$).

6.2.2.2 Chunk-Based Storage

Chunk-based storage is the next logical step from whole-file hashing. First, chunk boundaries are identified, and then whole chunks are stored uniquely, just like whole-file hashing. The algorithm to compute chunk divisions and their content addresses was described earlier in Section 4.4.3.

A file is stored in two parts: a list of the chunk identifiers and the chunk data. Chunks are stored as single instances of content-addressable objects. The fine-grained single-instance chunk storage improves on the efficiency of whole-file CAS when files are similar but not identical.

The shortcoming of chunking implementation are due to the tradeoffs that occur when fixing the parameters used to fulfill the requirement that the chunking algorithm be computed deterministically across all data. This is to ensure identical chunks can be found when new files are added to a system. Unfortunately, a fixed set of parameters does not optimally compress all data. Furthermore, chunk metadata (chunk lists) increases storage overhead which makes it less effective than whole-file CAS when unique files are stored.

Metrics Resemblance is piecewise, based on the new storage size.

$$\text{storage efficiency} = \frac{\sum_{i=1}^{k} r_i + q_i}{\sum_{i=1}^{k} v_i + q_i}$$

where for chunk $1 \leq i \leq k$, r_i is the size of a compressed and recorded, v_i is the size of of the uncompressed input source chunk, and q_i is the storage overhead.

6.2.2.3 Block-Based Storage

Fixed-size block-based storage, like Venti [125] is a degenerate case of chunk-based storage. Because the block division algorithm is trivial, it is easier to implement. It may also

benefit from underlying storage system efficiencies when the fixed block sizes are identical to a native file system block size. However, for read-only data, the variable-length block storage at the end of blocks is wasted, and insertions and deletions within files will prevent identical data from being detected.

Metrics Block-based storage is a degenerate case of chunk-based storage where the minimum and maximum chunk size is a constant.

6.2.3 Delta Compression Between Similar Files

When chunking indicates a stored file shows little or no resemblance to an existing file, or when the most space efficient method is desired, then delta compression should be used. Unlike chunking, it encodes differences between files at the granularity of a byte. A delta program like *xdelta* [95] or *vcdiff* [80] takes a *reference* and a *version* file as input, computes the difference, and outputs a *delta* file. The delta file contains **add** and **copy** operations which specify the bytes and region of bytes in the reference, respectively, that are output to form a reconstructed version file from the reference file.

The system first finds a reference file before running delta compression. The method used by our system computes a feature set for each reference and version file that is stored; we use Broder's definition of the *resemblance* metric, r in the range $0 \leq r \leq 1$ and by using feature (sub)sets, we estimate file similarity [12]. A feature set size, k in the range of 20 to 30 have been shown to be effective at finding similar files [40].

To compute a complete feature set for a file, we compute fingerprints over shingles, and then compute a feature vector from those shingles. This is described in Section 4.4.5.

The method for delta encoding resembling files is effective for eliminating redun-

dancy [40, 173]. Figure 6.2 is a graph of the efficiency of stored data, comparing *gzip* against *xdelta* (including *zlib*, comparable to *gzip* stream compression) as a percentage of the uncompressed files. The horizontal axis measures the discrete resemblance between a new file compared to a stored file with the highest resemblance. The vertical axis measures the size of all compressed data for a given resemblance against the original uncompressed data. The data set is the Linux kernel source code, versions 2.4.0-2.4.9, 88,322 files, 1.00 GB in size, $w = 30$, $k = 32$, and fingerprints are 32 bits long.

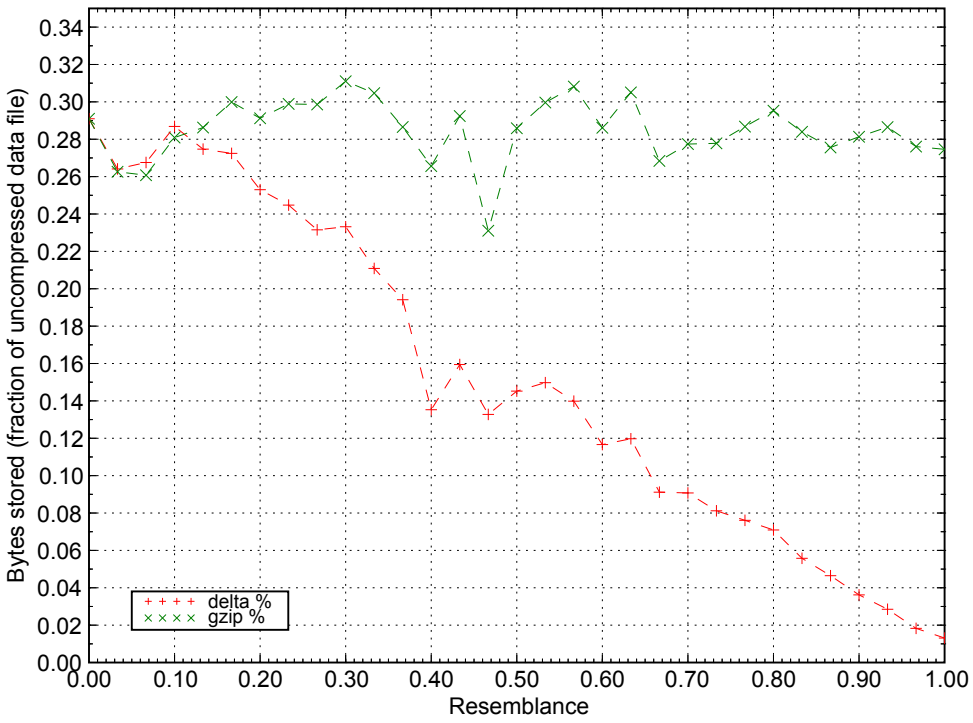


Figure 6.2: Storage efficiency of *xdelta* vs. *gzip*

We also ran the experiment with different sketch size, $k = 16$, and varied the window size, $w = \{8, 16, 32, 64, 128, 256\}$. The results, shown in Figure 6.3, show the different com-

pression rates. Better efficiency is exhibited at larger window sizes. However, the experimental results, reported earlier in Section 4.4.5, showed the highest compression rate (smallest size) at $w = 64$.

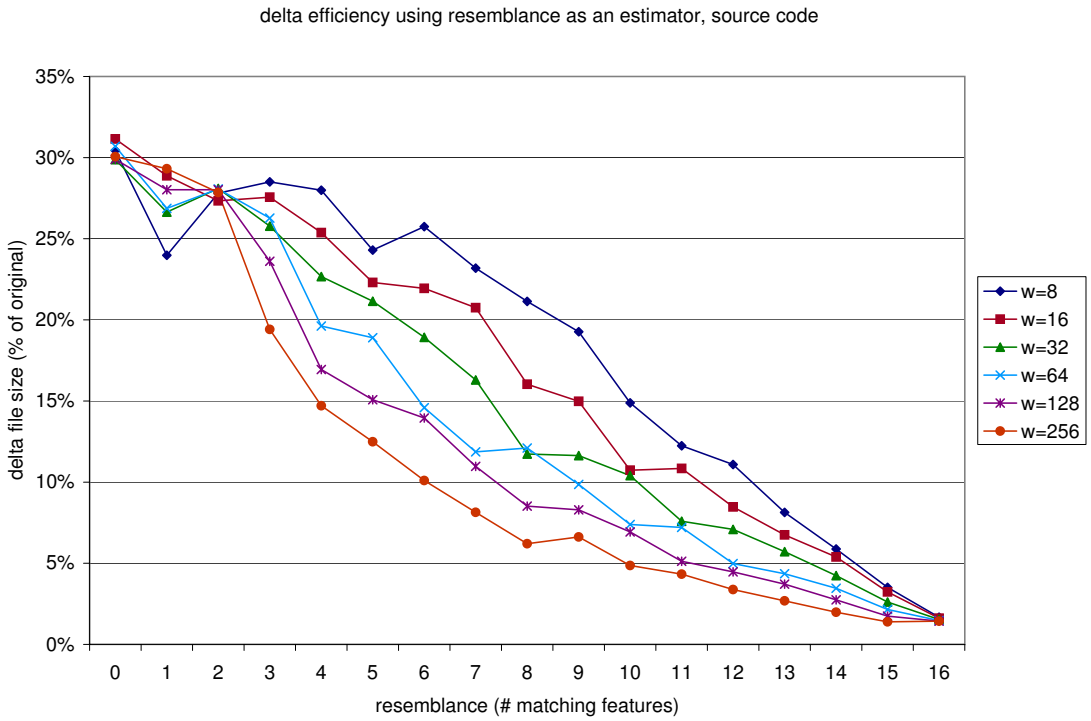


Figure 6.3: Storage efficiency for different shingle sizes

The delta compression algorithm is applied to one or more input files, resulting in a dependency graph. The one input file degenerate case is to compress a *version* file against an implicit "empty" file to compute an output *delta file* or *delta encoding*. The case of two input files *reference* file and an *version* file. (Although "version" might imply a direct relationship, in the general case no pre-existing relationship exists.) Multiple reference files with one version can be used also.

Figure 6.4 illustrates the relationship between **R**, the *reference file* (or *base file*), **V**,

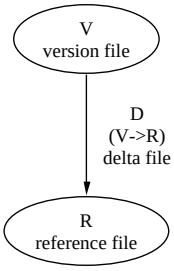


Figure 6.4: Reference, version, and delta files

the *version file*, and **D**, the *delta file*. The nodes represent files that are presented in the input. The edges represent data dependence, as well as a delta encoding file. These *delta files* are not presented to the application but are used internally as a data compression method.

There are two operations used in delta compression: computing a delta file from reference and version files, reconstructing a version file from reference and delta files. Note that in the general case, a version file may depend on more than one reference file. A version file may also be considered a degenerate case of an empty reference file. We use the notation $\mathbf{D}(\mathbf{V} \rightarrow \mathbf{R})$ to indicate the delta file **D** is an encoding and version file **V** depends on reference file **R**.

6.2.3.1 Selecting a Reference File

Delta compression computes the difference between a version file against one or more reference files; our implementation uses one. Files that are added to a system using the store operation can be modifications of an existing file, such as the extension of a system log, or they might be a modification from a common file, such as a document template.

In an ever expanding storage system, files will be added over time. Unlike systems that perform static analysis of the file set to determine similarity across all files [100, 40], DCSF

selects a reference file *incrementally*. To do this, we select the best reference file or object from which to compute a delta.

DCSF selects the best candidate reference file by using the following criteria in order:

- highest resemblance
- shortest delta chain
- highest degree of dependence

The *delta chain length* is computed at the version file. The dependency graph is a directed acyclic graph, so no cycles may occur and the length is the maximum of all possible paths from version to reference. In our experiments we restricted version files to depend on a single reference file; therefore the dependency graph is strictly a tree and the length is computed to the one reference file that does not depend on any other files. Figure 6.5 illustrates delta chain lengths. **R1**, a reference file, has length 0; it is not a version. The same is true for all **R***i*. **V2**, **V3a**, **V3b**, **V4a**, **V4b**, and **V4c** are all version files with chain length 1. **V3c**, **V4d**, and **V4e** all have length 2. Version files like **V3b** and **V4b** are used as reference files.

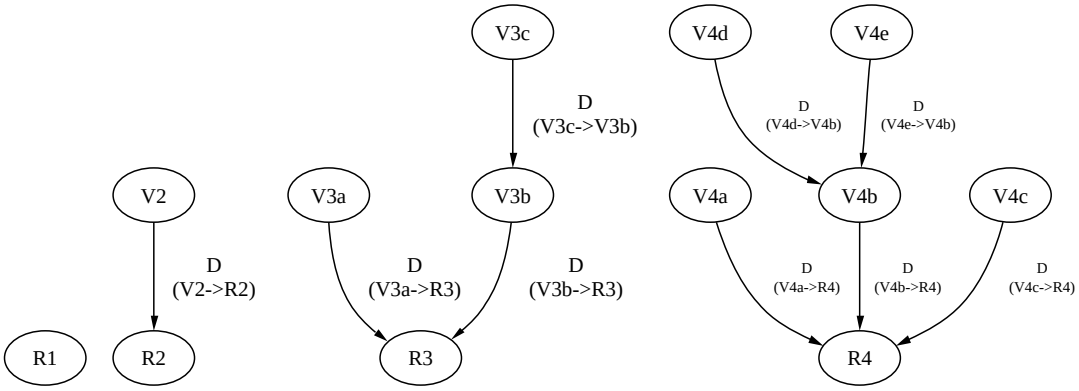


Figure 6.5: Delta chains and degrees of dependence

Another measure of data dependence is the *degree of dependence*, the total number of immediate version files that depend on the reference file itself. Version files can also be used as reference files. In Figure 6.5 it is simply the count of incoming edges coming into a node. No version files depend on **R1**, so its degree is 0; the same is true for **V2**, **V3a**, **V3c**, **V4a**, **V4d**, **V4c**, and **V4e**. **R2** has degree 1; **V2** depends on it. **R4**, a reference file, has degree 3 because version files **V4a**, **V4b**, and **V4c** all compute deltas against it.

File construction of any version file depends on its reference files. Since the reference files may also be version files, *e.g.* **V3b** and **V4b**, they must be reconstructed first.

6.2.3.2 Resemblance

Resemblance, an estimate of file similarity, is a metric between pairs of files. The delta operation forms a directed graph; shorter *delta chains* are beneficial. And the *degree of dependence* on a reference file elevates the importance of certain files. Resemblance [12], or the *Jaccard coefficient*, is computed as an overlap of documents by comparing shingles, or more precisely, resemblance r between two files A and B is defined as

$$r(A,B) = \frac{|S(A) \cap S(B)|}{|S(A) \cup S(B)|}$$

where $S(A)$ is a set of features. When two sketches are identical, $S(A) = S(B)$ then $r = 1$, and when no features in sketches $S(A)$ and $S(B)$ match then $r = 0$. We use ordered feature sets, or *feature vectors* so the resemblance between files A an B is discrete:

$$r(A,B) = \frac{|\{0 \leq i < k : f_i(A) = f_i(B)\}|}{k}$$

where $f_i(F)$ is a feature f of index i extracted from file F, and k is the size of the ordered sketch (array). In other words, each matching pair of elements in the feature vectors are compared

and the total matching features are taken as a fraction of the total number of features. The resemblance is in the range of $0 \leq r(A,B) \leq 1$.

6.2.4 Hybrid Methods

Hybrid storage methods are possible in PRESIDIO. For example, a large file might be first subdivided into chunks. The *progressive redundancy elimination*, or PRE, algorithm is then run on individual chunks, and the chunks can be stored as a delta representation on similar data instead of only suppressing identical storage. This approach is unique to PRESIDIO.

6.3 Evaluation

Our evaluation of resemblance detection and redundancy elimination consists of a series of experiments to show the effectiveness of different methods across a number of data sets. First, we measure and evaluate the relative differences between two different compression methods, namely *chunking* and *delta compression between similar files*. Next, we first measure the relationship between resemblance and inter-file compression using delta compression.

6.3.1 Comparing Chunking and Delta Compression

We made the first direct comparison of the two main compression techniques proposed in the literature, namely chunking and delta compression, and compare them against intra-file compression. In general, both chunking and delta encoding outperform *gzip*, especially when they are combined with compression of individual chunks and deltas.

As noted above, storage efficiency is the determining factor for the applicability of an inter-file compression approach. We report on the storage space required as a percentage of the original, uncompressed data set size. For example, stored data that is 20% the size of the orig-

inal represents an efficiency ratio of 5:1. The functionality and performance of each approach depends on the settings of a number of parameters. As expected, experimental results indicate that no single parameter setting provides optimal results for all data sets. Thus, we first report on parameter tuning for each approach and different data sets. Then, using optimal parameters for each data set, we compare the overall storage efficiency achieved by each approach. The required storage includes the overhead due to the metadata that needs to be stored. Last, we discuss the performance cost and the design issues of applying the two techniques to an archival storage system.

One problem that comes up in the configuration of the chunking algorithm is setting parameters that maximize storage efficiency. The experimental chunking parameters are listed in Table 4.2 and represented graphically in Figure 4.3. The size of the hash identifiers computed from the identifying (strong hash) algorithm increases the per-chunk storage. The original files can be reconstructed from their constituent chunks. To do that, the system needs to maintain metadata that maps file identifiers to a list of chunk identifiers. Furthermore, the chunk size (a distribution) is inversely proportional to the chunk list overhead.

6.3.2 The *chc* Program

In order to measure chunking efficiency before we were able to construct a content-addressable store, we wrote the *chc* program which compresses an input file to produce a "chunk compressed" output file (`.chc` suffix); a separate option performs the inverse operation to reconstruct the original input file. The *chc* program emits statistical information (chunk sizes, sharing) and the `.chc` file contains the necessary overhead used to evaluate the experimental data using the metrics we desired.

The file format shown in Figure 6.6 consists of the following: a list of chunk IDs that

identify the chunks, in order, from the original input file; and a sequence of chunkIDs, the size of the chunk, and the variable-length chunk data.

The separate step of compressing the chunk file was performed by executing *gzip*, passing the `.chc` file as input.

We validated *chc* by computing the intersection of chunk IDs from a `.tar` file and from chunk IDs of individual files using the UCSC SSRC web directory (122.8 MB, 721 files, $d = 1024, m = 64, M = 1024$) and 99% of the chunks detected in the *tar* file were identical to the chunks in the individual files.

An optimistic estimate of the storage overhead comprises of a list of chunk identifiers for each file, and for each chunk, the chunk identifier and the chunk size. A production system might incur additional overhead from managing variable-sized blocks, in-memory or on-disk hash tables, as well as file metadata.

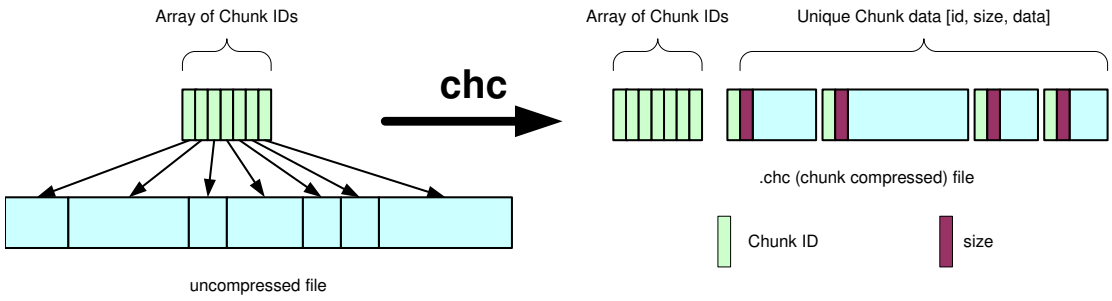


Figure 6.6: The *chc* program file format

The data objects referenced were simply the stored chunks. Each data set was combined into a single *tar* file and then the file was divided into chunks. (The use of the *tar* file was used as an approximation to chunking each file individually.) Single instances of chunks were stored using the parameters listed in Table 6.1.

Every chunk identified in the *tar* file was sent to the CAS to be stored. The first time

polynomial	(1)a8948691
divisor	1,024
residual	0
min chunk	64
max chunk	16,384
windowsize	32

Table 6.1: *chc* parameters

	delta/shingle parameter
w	shingle window size (24 bytes)
k	degree of shingle fingerprint (32 bits)
$p(x)$	random irreducible polynomial for window
s	feature set size (30)
t	delta threshold (1)
l	degree of feature fingerprint (32 bits)
L	maximum length of a delta chain
$q_i(x)$	random irreducible polynomial for window

Table 6.2: *dcsf* parameters

a chunk was stored, its reference count was set to 1; subsequent store requests were suppressed but the reference count was incremented. Finally, the CAS tabulated the reference counts over all chunks.

6.3.3 Delta Encoding Similar Files

Delta encoding tools generally do not have any parameters even though there can be algorithmic variations. Detecting similarity requires a number of parameters, as shown in Table 6.2.

The fingerprint of a window (of size w) is represented as $RF(A_n)$ using the following expressions:

$A(x)$, binary string representing the file

$A_0(x)$, binary string starting at byte 0, *i.e.* $A[0..windowsize-1]$

$A_1(x)$, binary string starting at byte 1, *i.e.* $A[1..windowsize]$

$p(x)$, the random irreducible polynomial of degree k

$RF(A_0) = B_0(x) \bmod p(x)$: fingerprint of the window (degree k)

Next, the feature set of size s is selected by computing fingerprints of the window fingerprints as follows: An array containing s tuples will contain the following: 1. a random irreducible polynomial $q_i(x)$ of degree l, 2. an input string $feature_i$ of size k and 3. the output Rabin fingerprint $featureprint_i$ of degree l.

---

for $i = 0$ to $i < s$ **do**

$featureprint_i(feature_i) = feature_i(x) \bmod q_i(x)$

end for

Each element in the array represents a different fingerprinting function for selecting one feature. Each window fingerprint $RF(A_n)$ that is computed is used as input into the s fingerprinting functions in the array. The input is selected by minimizing the output of the fingerprint, *i.e.* when each fingerprint $featureprint_i$ is computed with the $feature_i$ set to $RF(A_n)$, the output is compared against the previously retained output and if it is lower, the new $feature_i$ and $featureprint_i$ are retained. Comparing two sketches using the *approximate min-wise independent permutations* [14] is evaluated by comparing corresponding elements in the two sketch arrays.

6.3.4 Delta and Resemblance Parameters

Each sketch of s fingerprints $feature_i$ of degree l, for example l = 32 bits, is stored, for a total of $4s$ bytes.

We choose two parameters, the feature set size, $s = 30$, and the window size, $w = 24$,

within ranges that have been shown to provide meaningful resemblance metrics—and more importantly—a strong (inverse) correlation with the size of the delta encoding [40].

In order to determine whether a new file should be added as a new *reference* file or a *version* (delta encoded) file, a resemblance is computed using a pairwise comparison of features between two sketches. A delta encoding is computed if the number of matches is greater than the t threshold (1).

The third problem is about calculating the actual delta, when a reference file within the resemblance threshold is found. This is a well-explored area, with a number of available tools, such as *xdelta*, *zdelta* and *vcdiff*. We used *xdelta* for this experiment. A pointer to the reference file has to be maintained with every delta in the system. Such identifiers (*e.g.* , SHA digests) as well as file sketches contribute to some storage overhead that has to be taken into account. Again, a delta can be further compressed by using *gzip*. Our prototype consists of three programs, one for each of the three above problems: feature extraction, resemblance detection, and delta generation.

6.3.5 Data Sets

To establish a baseline for each data set, we created a single *tar* file from the data set, and then compressed it with an intra-file compression program, *gzip*. As expected (and as shown by the two first rows of the table), inter-file compression improves with larger corpus sizes. This is not the case with *gzip*.

- **HP Support Unix Logs** Text, computer generated, multiple hosts, different hardware, different network locations. Highly similar data.
- **Linux Kernel 2.2 Source** Four separate versions (2.2.0, .7, .14, and .25) of the Linux kernel source code. Text, human generated, versions of the same files.

Data Set	Size	# Files	tar & gzip	Chunk	Chunk & zlib	Delta	Delta & zlib
HP Unix Logs	824 MB	500	15%	13%	5.0%	3.0%	1.0%
HP Unix Logs	13,664 MB	8,000	14%	11%	7.7%	4.0%	0.94%
Linux 2.2 source (4 vers.)	255 MB	20,400	23%	57%	22%	44%	24%
Email (single user)	549 MB	544	52%	98%	62%	84%	50%
Mailing List (BLU)	45 MB	46	22%	98%	53%	67%	21%
HP ITRC Web Pages	71 MB	4,751	16%	86%	33%	50%	26%
PowerPoint	14 MB	19	67%	55%	46%	38%	31%
Digital raster graphics	430 MB	83	42%	102%	55%	99%	42%

Table 6.3: Comparison of storage efficiency from different compression methods

- **HP ITRC Support Web Site** Text, human generated with some computer-generated markup, some support page similarity.
- **Microsoft PowerPoint Presentations** Multiple presentations that have evolved over time on related topics. Binary, human generated text plus PowerPoint art and some bitmap imagery.
- **Imaging DRG** California Digital Raster Graphics. Binary/bitmap, stored in non-lossy format, scanned from topographical maps. [27]
- **Email** Personal (single-user) email folder. Text, human generated.
- **Mailing Lists** Email mailing list archives. Text, human generated. [Kraftwerk music interest, ImageMagick developer list, and BLU-Discuss (Boston Linux and Unix User Group)].

The HP Unix Logs (8,000 files) show very high similarity. Chunk-based compression on this similar data was reduced to 11% of the original data size, and when each chunk is

compressed using the *zlib* (similar to *gzip*) compression, it is just 7.7% of the original size. Even more impressive are the reductions in size when using delta compression. When delta compression is used alone, the data set is reduced to 4% of the original size, but when combined with *zlib* compression, the compressed data is less than 1% of the original size.

Textual content, such as web pages, can be highly similar. However, in the case of the HP ITRC content, *gzip* compression is more efficient than chunking or delta. More surprisingly, *gzip* is better even when we do additional compression of chunks and deltas. The reason is that *gzip*'s dictionary is more efficient across entire files than within the smaller individual chunks, and chunk IDs appear as random (essentially non-compressible) data. But in the context of an archival storage system, *gzip*'s advantage is not likely to be as effective in practice; this is discussed below.

Non-textual data, such as the PowerPoint files with chunking and delta (especially with *gzip*) achieve better efficiency than *gzip* alone. However, the achieved compression rates are less impressive than those for the log data. For raster graphics, delta encoding with *gzip* achieves modest improvement over *gzip* alone.

A single user's email directory and a mailing list archive show little improvement when using delta. Chunking is less effective than *gzip*, although we would expect it to reduce redundancy found across multiple users' data.

In most cases, inter-file compression outperforms intra-file compression, especially when individual chunks and deltas are internally compressed. Chunking achieves impressive efficiency for large volumes of very similar data. On the other hand, delta encoding seems better for less similar data. We believe that this is due to the lower storage overhead required for delta metadata. Typical sketch sizes of 80 to 120 bytes (20 to 30 features $\times$ 4 bytes) for a file of any size are significantly smaller than the overhead of chunk-based storage, which is linear

with the size of the file.

Although compressing a set of files into a single *gzip* file to establish a baseline measurement helps illustrate how much redundancy might exist within a data set, it is not likely that an archival storage system would reach those levels of efficiency for several reasons. Most important is that files would be added to an archival system over time and files would be retrieved individually. If a new file were added to the archival store, it would not be stored as efficiently unless the file could be incorporated into an existing compressed file collection, *i.e.* the new file would need to be added to an existing *tar*/*gzip* file. Likewise, retrieving a file would require first extracting it from a compressed collection and this would require additional time and resources over a chunk or delta-based file retrieval method.

Our experiments measured the size of an entire corpus, in the form of a *tar* file after it has been compressed with *gzip*. Had we compressed each file with *gzip* first and then computed the aggregate size of all compressed files, the sizes for *gzip*-compressed files would have been much larger. For example, in the case of the HP ITRC web pages, *gzip* efficiency would have been 30% of the original size, much larger than the 16% shown in table 6.3, and larger than the 26% that can be achieved by using delta compression with *zlib*. When delta compression (or to a lesser extent, chunking) is applied across files first and then an intra-file compression method second, it is more effective than compressing large collections of data because additional redundancy can be eliminated.

Chunking Parameters Chunking efficiency varies with the different chunking parameters. Chunking, which divides and addresses blocks of data using hashing techniques, relies on determinism for stateless and consistent distributed behavior: data that is chunked by one host does not need to see the data from another host in order for common chunks to be identified.

This constraint imposes the restriction that the parameters for chunking must be determined once, and for all, data that is intended to be stored efficiently.

In Figure 6.7, we see that universally optimal parameters do not exist. In the case of highly similar data, "logs-10", a 10 file subset of the HP Log data, chunking efficiency improves with smaller chunk sizes, which is determined by the *divisor* parameter. The irregular curvature of the graph is due to effect of chunking overhead. The figure also shows that when similarity is low, compression efficiency is poor and the overhead can increase the storage requirements above the size of the original corpus.

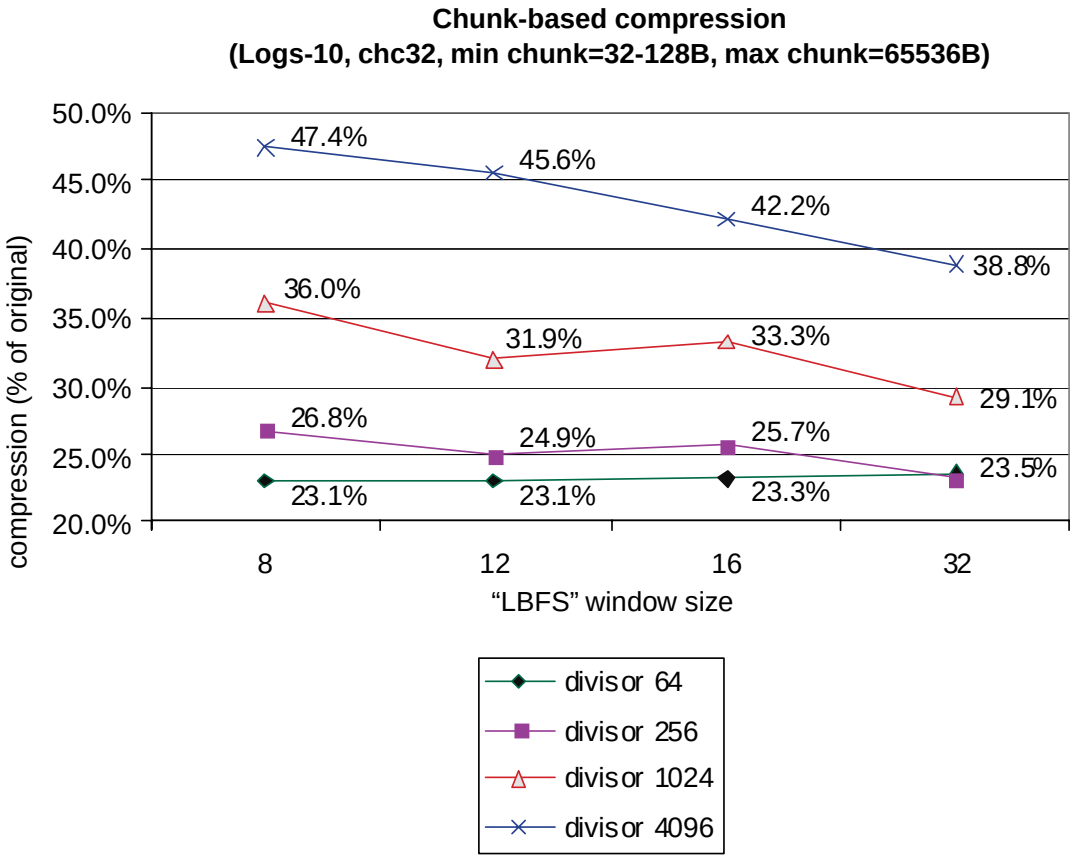


Figure 6.7: The space efficiency of chunk-based compression at different window and expected chunk sizes

The chunking parameters for the effect of the window size on chunking is shown in Figure 6.8. With larger chunk sizes, larger window sizes, w can improve storage efficiency with no additional cost. In this example of the text-based "log" data, a window size of 128 bytes provided the best efficiency (35.7%) for an expected chunk size of 4096 bytes (d = 4096).

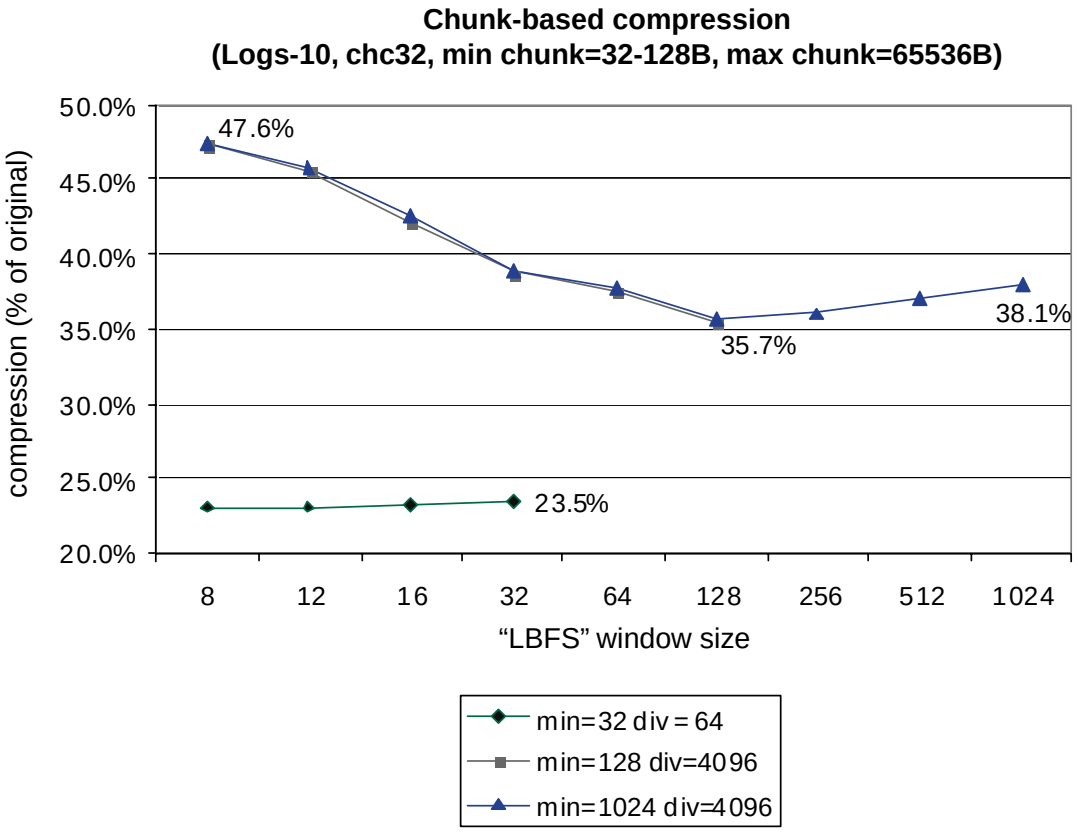


Figure 6.8: The space efficiency of chunk-based compression at different window sizes

Chunking Overhead In the case of chunking, the *expected chunk size* is a key configuration parameter. It is implicitly set by setting the fingerprint divisor as well as the minimum and maximum allowed chunk size. In general, the smaller it is, the higher the probability of detecting common chunks among files. For data with very high inter-file similarity (such as log files),

small chunk sizes result in greater storage efficiency. However, for most data this is not the case, because smaller chunks also mean higher metadata overhead. Often, because of this overhead the storage space required may be greater than the size of the original corpus. As Figure 6.9 shows, the optimal expected chunk size depends on the type of data; using 128-bit identifiers, the best efficiencies range from 256 to 512 bytes.

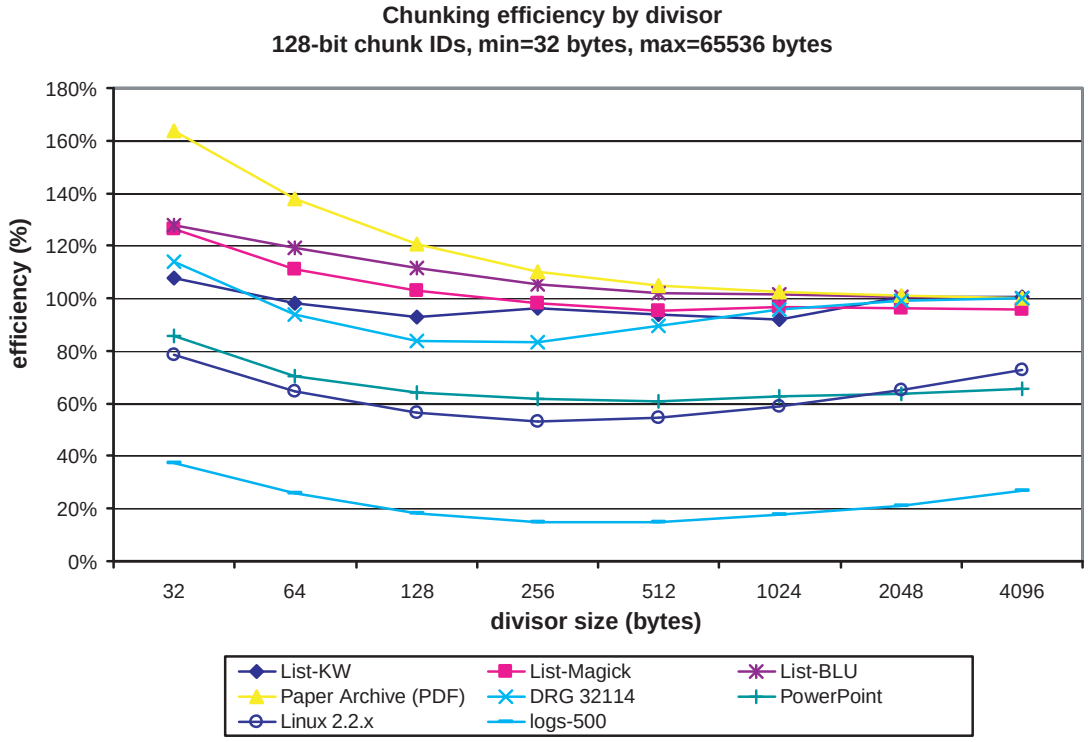


Figure 6.9: Chunking efficiency by divisor size

Comparing Similarity and Chunking Overhead For storage efficiency, delta encoded data works well at both ends of the spectrum: when data is highly similar, the total efficiency of delta-encoded storage is nearly the same as chunking when both are combined with *gzip* compression. But for data that is less similar, or even completely dissimilar, delta encoded data

does not exhibit nearly as much overhead since a sketch of 120 bytes for a file of any size is significantly smaller than the overhead of chunk-based storage which is linear with the size of the file. (Furthermore, with a small loss of efficiency, a sketch size of 80 bytes is nearly as effective [40].)

For example, chunks that have an expected size of 1024 bytes (chunking parameter d), a chunk ID size (CID_{size}) of 128 bits will incur approximately $(filesize/1024) \times 16$ bytes, plus additional overhead for storing the chunks themselves of 20 bytes.

6.4 Evaluating Methods for Improving Compression

6.4.1 Delta compression between similar files

Delta encoding [4], an excellent inter-file compression method, is a compelling mechanism for reducing storage within a corpus that stores similar files. Delta compression is used to compute a delta encoding between a new *version file* and a *reference file* already stored in the system. When resemblance, an estimate of the similarity between two files, is above a predetermined threshold, a delta is calculated and only that is stored in the system. There are three key steps that need to be designed and implemented for efficiency to delta compress between similar files (DCSF).

First, features are selected from files in a content-independent and efficient way. We use the shingling technique (DERD) by Douglis *et al.* [40], which calculates Rabin fingerprints [11] over a sliding window on byte boundaries along an entire file. The window size, w is a preselected parameter. The number of intermediate fingerprints produced is proportional to the file size. To reduce it to a manageable size, a deterministic *feature selection* algorithm selects a fixed size (k) subset of those fingerprints (using *approximate min-wise independent*

permutations [14]) into a *sketch*, which is retained and later used to compute an estimate of the *resemblance* between two files by comparing two sketches. This estimate of similarity is computed between two files by counting the number of matching pairs of features between two sketches. Douglis has shown that even small sketches, *e.g.* sets of 20 features, capture sufficient degrees of resemblance. Our experiments also show sketch sizes between 16 and 64 features using 32-bit fingerprints to produce nearly identical compression efficiency. Using the same hardware as above, we measured our feature selection program at 19.7 MB/s, $k = 16$, reading a 100 MB input file.

Second, when new data needs to be stored, the system finds an appropriate reference file in the system: a file exhibiting a high degree of resemblance with the new data. In general, this is a computationally intensive task (especially given the expected size of archival data repositories). We are currently investigating feature-based hashing to reduce the search complexity. Our method differs from DERD by allowing delta chains of length greater than one, by storing and detecting similar files incrementally to more closely match a growing archive. We used sketch sizes of 16 features ($k = 16$) and sliding window size of 24 bytes ($w = 24$).

Third, deltas are computed between similar files. Storage efficiency from delta is directly related to the resemblance. Highly similar files are more effective at reducing storage usage than mildly similar ones. Fortunately, it is possible to reduce the comparison between pairs of fingerprints in feature sets (16 or more fingerprints each) down to a smaller number of features that are combined into "super" features, or superfingerprints/supershingles [15]. Each superfingerprint is a hash of a subset of the features. If one or more superfingprints match, there is high probability of a high similarity. REBL [83], by Kulkarni *et al.* , computed superfingerprints over chunks with 1 KB to 4 KB average size to yield large numbers of superfingerprints to detect files with high resemblance. Because our design constraints such as the number of

expected files and real memory prevent this method from being directly applicable, we are evaluating a solution to meet those requirements. We measured *xdelta* at 8.7 MB/s (20 MB total input size, 2.29 seconds) for worst case performance: a 10 MB input file of zeros and a 10 MB file of random data, producing a 10 MB output file. Delta compression programs with higher performance are known to exist [162] and can be used in practice.

The fourth step, common to all efficient storage methods, is storing the compressed data. In DCSF, the *delta file* is recorded to the CAS.

6.4.2 Measurements

To evaluate our expected storage efficiency, we compressed six data sets using stream compression (*gzip*), chunking (*chc32*), and delta compression between similar files (*dcsf*), measuring the total (not just incremental) compressed size. Figure 6.10 shows these measurements as a percentage of the original data.

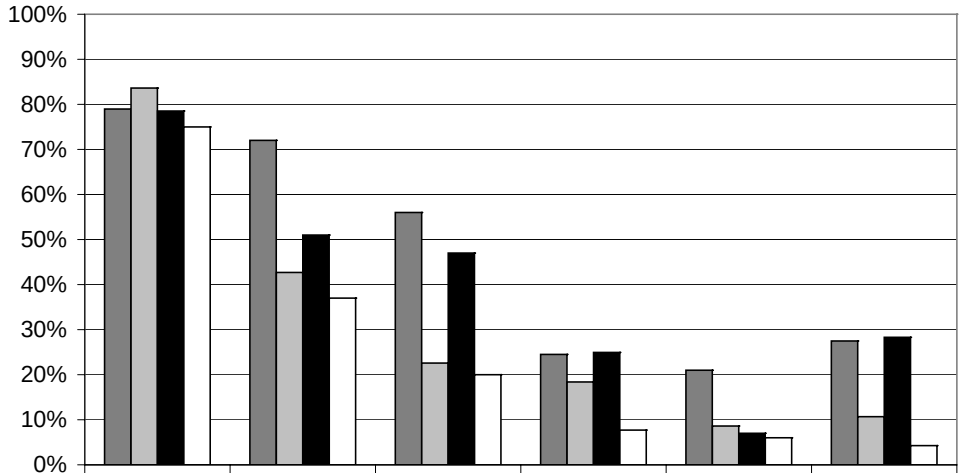
To illustrate the range of redundancy in data, we selected data sets that are likely to be archived, binary and textual data, and small and large files, and dissimilar as well as highly similar data. Table 6.4 lists for each set its total size in megabytes, number of files, and average file size in bytes.

Name	**size (MB)**	**# files**	**avg. size (B)**	**s.d**
PDF papers	239	754	331,908	802,035
PPT presentations	63	91	722,261	944,486
Mail mbox (3 dates)	837	383	2,291,208	9,397,150
HTML	545	40,000	14,276	27,317
PDF statements	14	77	186,401	120,146
Linux 2.4.0-9	1,028	88,323	12,209	31,528

Table 6.4: Data sets

We briefly describe the content in these data sets. *PDF papers* are PDF documents

of technical papers (conference proceedings, journal articles, and technical reports) in a bibliographic research archive. *PPT presentations* are Microsoft PowerPoint presentation files of related work. *Mail mbox* are email folder snapshots for a single user on three separate dates. The *HTML* set comes from the *zdelta* benchmark [162]. *PDF statements* are monthly financial statements from investment institutions. And *Linux 2.4.0-9* is ten versions of the Linux kernel source code, 2.4.0 through 2.4.9.



	PDF papers	PPT presentations	Mail mbox (3 dates)	HTML (40,000)	PDF statements	Linux 2.4.0-9
gzip each	79.0%	72.0%	56.0%	24.5%	21.0%	27.5%
chc32 -d 512 -z	83.6%	42.7%	22.6%	18.4%	8.6%	10.7%
dcsf -L 1	78.5%	51.0%	47.0%	24.9%	7.0%	28.3%
dcsf	75.0%	37.0%	20.0%	7.7%	6.0%	4.3%

Figure 6.10: Storage efficiency by method

Sizes were measured in the following manner. The *gzip* compressor was applied on each file with default parameters, and all file sizes were added to produce the compressed size. The *chc* program read a *tar* file containing all input files and produced a single chunk archive, using a divisor of 512 bytes ($D = 512$), compressing each chunk with the *zlib* stream

compressor; the total size is a sum of the single instances of compressed chunks and a chunk list. The *dcsf* method computed delta using *xdelta* (version 1.1.3 with standard options that use the *zlib* compressor), selecting the best file with a threshold of at least one matching fingerprint in the sketch; reference and non-matching files were compressed with *gzip* and the measured size was the sum of all of these files. The `-L 1` option sets a maximum delta chain length of one, *i.e.* deltas are only computed against reference files. This avoids chains of reconstruction, but at the expense of lower space efficiency.

6.4.3 Measuring the benefit of high resemblance data

To help understand the importance of file similarity on the storage efficiency, we conducted experiments to compute the relationship between resemblance (an estimate of file similarity) and the actual amount of storage that is used. The amount of data similarity varied widely, and so did the amount of compression. However, some behaviors were common, such as the relationship of storage resemblance and the inter-file compressibility of data. Using the Linux source code data set (ten versions, 2.4.0–2.4.9), we ran *dcsf* to assess the importance of high resemblance data. Our experiments would store all files from version 2.4.0 first in order, then 2.4.1, etc. evaluating each file individually against previously stored files. The file size after *gzip* (intra-file compression only) was compared against *xdelta* (both intra- and inter-file compression) and total storage was tabulated by resemblance.

In Figure 6.2, the horizontal axis lists the (discrete) resemblance: a number of features that were matched and for each corresponding number of matching features (out of a sketch size $k = 30$). On the vertical axis, the amount of space that was necessary to store the files with that resemblance. The *delta* graph shows the storage efficiency improving (decreasing) as the resemblance increases. This confirms the relationship between resemblance (an estimate) and delta (a

computed difference between two files). By comparison, the *gzip* graph is relatively flat, ranging from approximately 25% to 30%. The reduction in storage illustrates the complementary benefit of intra-file and inter-file compression.

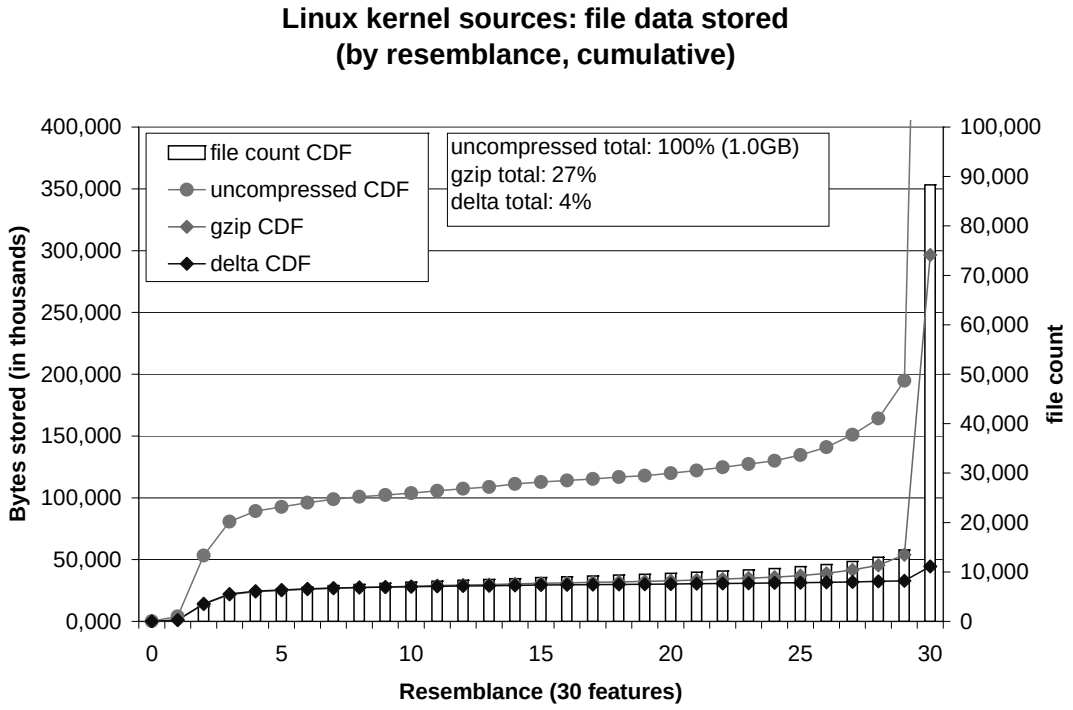


Figure 6.11: Cumulative data stored, by resemblance comparing *gzip* and delta compression; 1.00 GB cumulative uncompressed size

Using the same data, Figure 6.11 shows a different view of the storage efficiency that demonstrates the importance of finding identical or highly similar data. The resemblance is still on the horizontal axis, but two sets of data are superimposed. The file count (bar graph) shows the number of files that are in the workload with a given resemblance. The other lines show both uncompressed size, the size of the data set when each file is compressed with *gzip*, and finally the delta-compressed size using *xdelta*. File counts and sizes are cumulative of all files with lower resemblance.

With 88,323 files and one gigabyte of data, a significant number of files have very high similarity, in fact many are identical. The amount of storage required for *gzip* is only 27% but with *delta*, the total amount of storage is 4% of the original, uncompressed source code. The relative flatness of the *delta* plot near 30 of 30 features shows only a slight increase in storage space despite the large numbers of copies that were stored.

What is important to note is that the major benefit comes from files that have high resemblance (30 out of 30 matching features, which is equivalent to all superfingerprints matching). PRESIDIO's progressive feature matching process would first attempt to match identical files, then highly similar files, and then finally somewhat similar files. The time to search the first two categories is relatively fast and requires direct lookup of whole files or chunks instead of a full pair-wise feature resemblance across a large set of files.

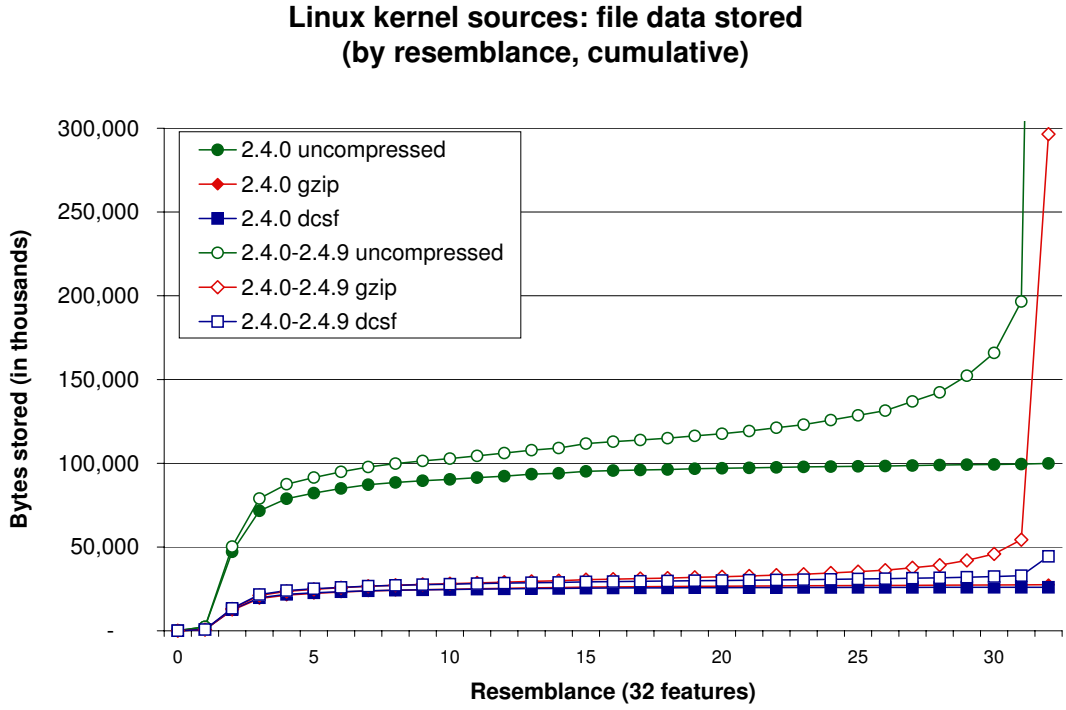


Figure 6.12: Cumulative data , by resemblance; Linux 2.4.0 vs. 2.4.0–2.4.10; 1.00 GB cumulative uncompressed size

method	bytes		size of 2.4.0	% of uncompressed	
	2.4.0	2.4.0–2.4.9	vs. 2.4.0–2.4.9	2.4.0	2.4.0–2.4.9
uncompressed	99,944,270	1,078,316,273	9.27%	100.00%	100.00%
gzip	27,527,744	296,475,233	9.29%	27.54%	27.49%
dcsf	26,005,285	44,446,197	58.51%	26.02%	4.12%

Table 6.5: Cumulative data stored by dcsf, one version of source versus ten versions

Another experiment compares DCSF on a single set of source code, Linux 2.4.0, against DCSF on ten sets of source code, versions 2.4.0 through 2.4.9. This experiment used a sketch size of $k = 32$ and produced nearly identical compression results to the previous experiment $k = 30$ for ten versions. The results are shown in Table 6.5. As would be expected of incremental source code development, the uncompressed data size for one version is 9.27% or about one-tenth of ten versions. Likewise, the total size of all files in a single version individually compressed by *gzip* is 9.29%, also close to one-tenth of all ten versions of source files gzipped individually.

This experiment reveals discriminating behavior when DCSF is applied to both dissimilar and highly similar data. While a single version of source files are individually compressed with *gzip* to 27.54% of their total uncompressed size, DCSF compresses slightly better, to 26.02% of total size. The slight improvement is due to the slight favor of delta compression to *gzip* between files with non-zero resemblance. The more dramatic result is that storing ten versions of source requires 44.4 million bytes versus 26.0 million bytes—an increase of 70.91% to store an additional 979% more data.

Figure 6.12 illustrates the the effect of resemblance on compression. Each file is computed to have a resemblance to at least one other file previously stored. The height of a data point represents the cumulative amount of data stored for all files less than or equal to the resemblance on the x-axis. Solid data points indicate a single version; outlined data points

are for ten versions. The shape of uncompressed and *gzip*-compressed curves are similar, but reduced by a nearly constant factor for *gzip*. The difference between uncompressed (circles) and DCSF (squares) is very different for all ten (2.4.0–2.4.9) versions of source: most data in uncompressed form is highly similar, as indicated by the data point at 1,078,316,273 bytes, off the chart. However, for both single-version and ten-version curves, the additional storage require to store highly similar data is very small due to the high efficiency of delta compression.

By experimenting with DCSF and varied parameters, we have measured a number of data sets which have high variation in the amount of inter-file and intra-file redundancy. High levels of compression are possible, including computer-generated data that was measured to less than 1% of the original size [173]. By not attempting to fit a single compression scheme to all data, and providing a framework for one or more schemes, the Deep Store architecture benefits a wide range of data types.

6.4.3.1 Delta Chain Length

Delta compression creates dependencies between files. These dependencies form graphs that affect the size of stored data, storage and retrieval performance, and reliability. It is important to select shorter reference files with delta chains.The cost of retrieving a file by reconstructing data from reference and delta files is directly related to the length of the delta chains. It is important to select an appropriate delta chain length. We have measured the effect of short and long chain lengths.

Delta chains of length greater than one benefit compression significantly. Previous work using delta compression to store data efficiently have not used delta chains [40, 83] and only used a single delta file to compress against a reference file and not another version file, *i.e.* chain length $L \leq 1$. The measurements listed in figure 6.10 show the potential difference

between bounded (`dcsf -L 1`) and unbounded (`dcsf`) delta chains. When the maximum chain length is unbounded, storage can be reduced by nearly seven times down to a storage rate of 4.3% (in the case of Linux 2.4.0–2.4.9).

Figure 6.13 shows the cumulative distribution of files against the degree of dependence on base files. Each data set is graphed twice, once using delta chains of maximum length 1, *e.g.* `html -l -r -L 1` in dashed lines, and again using delta chains of unbounded length. Limiting the delta chain length also limits the degree of dependence. Higher numbers of files with lower degrees of dependence (L 1) indicate lower instances of delta compression, which is in turn reflected by lower storage efficiency. In contrast, unbounded delta chains allow higher levels of data dependence. One reason for this effect is that files are introduced into the system one at a time.

The Linux data set contains ten similar versions of source code. The lowered data points below 95% for degrees less than 9 indicate the high level of delta encoding against similar files, including high degrees of dependence as high as 8. One can picture a single file (version 2.4.0) being introduced to the system, followed 9 identical or similar files being detected (2.4.1 through 2.4.9). Delta files are computed against the original.

But this restriction is a simplifying assumption and relaxing it to allow arbitrary length increases storage efficiently significantly. This is in part due to the problem that a file can only be a reference file or version file. Once a file is committed to storage as a version file (*i.e.* the file is considered a version file and a delta is stored in its place), it can no longer be used as a reference. A low resemblance threshold compounds this problem by allowing more files to be considered as versions of existing reference files than when a high threshold is used. When no *a priori* knowledge of future files is known, highly similar files might become version files. Our measurements show this is true, rendering a highly similar set of file data to compress no better

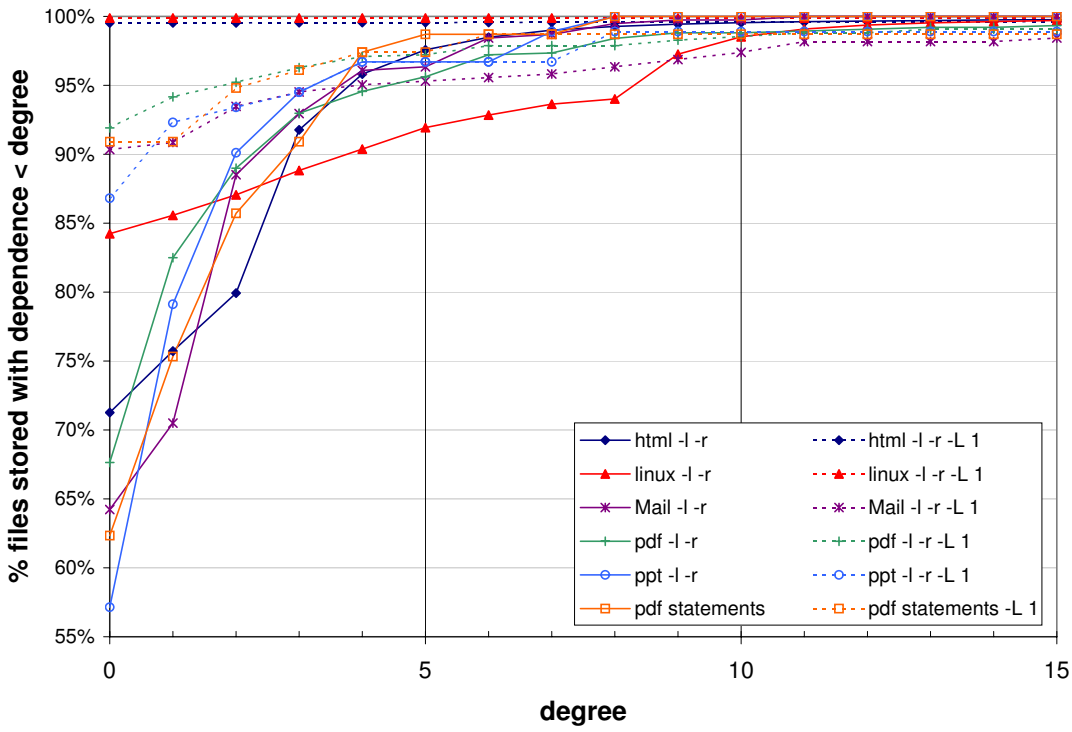


Figure 6.13: Cumulative distribution of files stored by degree of dependence

than *gzip* on a per-file basis.

Delta chains can be detrimental to storage performance and resource usage. Reference files can be virtual, meaning that they must first be reconstructed. When reference files, or version files that are used as delta references, must be retrieved or reconstructed, disk and computation are used first to reassemble and then store the fully instantiated reference file.

We mitigate reconstruction cost from delta chains in a number of ways. The first and most important step is to reduce the delta chains when it is possible. If the reference file is selected arbitrarily, long delta chains can form. A simple change to associate a single delta chain length remedies the situation significantly, as seen in Figure 6.14. In this example, the reference

file with the highest resemblance is selected. The first line listed in the legend minimizes the delta chain in selection; all things being equal, a reference file with the lowest delta chain length is selected. This resulted in an average chain length of 5.83. The second line attempts to compute delta against the maximum degree, but produced an average chain length over double (12.91) of the previous method. The third method selected an arbitrary reference file, with an average chain length of 24.26. The last case is the same as the first, with the exception that the maximum chain length was 8 (-L 8). This may be useful for analysis of reliability models at a small expense of storage space. With high levels of resemblance seen in this data set, careful base file selection can reduce chain length or increase degree of dependence with low cost. The difference between minimum and maximum compression between these methods is approximately 3%, which is negligible compared to the 23.5:1 compression ratio.

A possible improvement is to rewrite some version files as reference files. When the system detects a large degree of dependence on a version file, the version file can be retrieved and stored as a reference.

Caching might further mitigate the effect of delta chains. The issues in delta chains are largely related to performance, so file caching techniques, including predictive prefetching, may be useful. To date, we know of no research that has been conducted on caching in delta compressed archival storage systems.

6.4.3.2 Data Dependent Compression

We measured the storage efficiency of *dcsf* over a number of data sets. In Figure 6.15 we show the amount of storage required to compress several data sets. The vertical axis indicates the total amount of space used when delta compression is applied above a threshold, indicated on the horizontal axis. (For illustration, last data point on the data graph indicates the cumulative

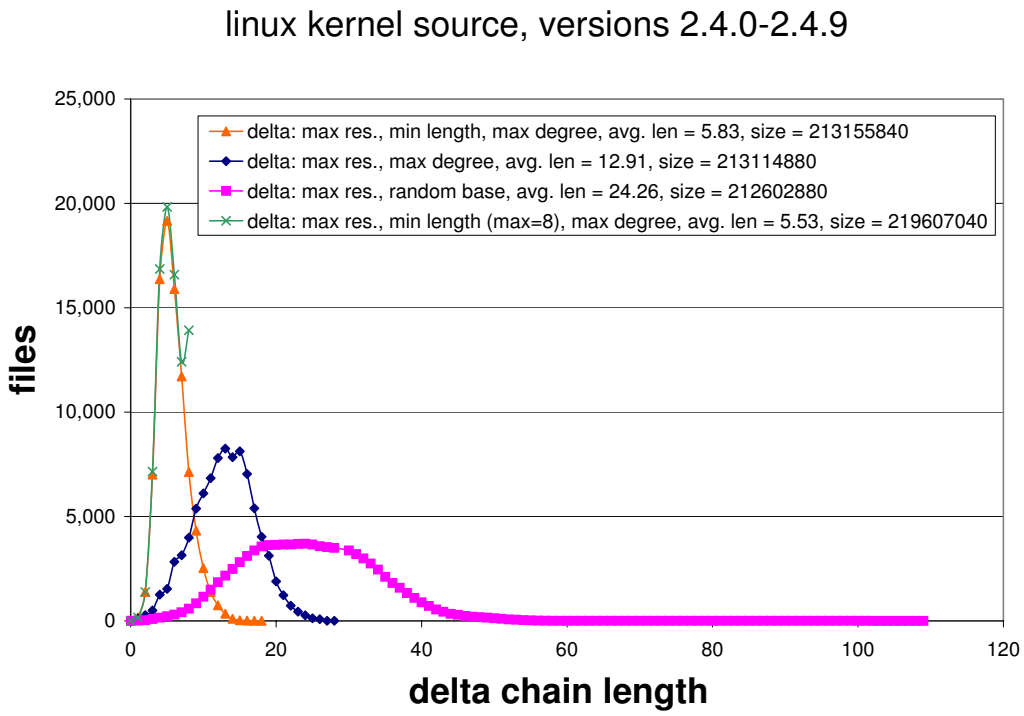


Figure 6.14: Delta chain length and storage efficiency

storage when each file is compressed with *gzip*.) For example, the Linux source data set (ten versions) shows on the far right that if delta compression were applied to files showing perfect resemblance of two sketches (at $r = 1.00$, equivalent to a single superfingerprint when $s = 1$), the compression reduces storage to 6% of the total, a saving which would be realized by one supershingle. The data set labeled Statements would be reduced to 14.8% of uncompressed data ($r = 1.00$), but more significantly, down to 2.5% ($r = 0.77$). With high probability, the harmonic superfingerprints at $s = 8$ would be able to detect those matching pairs. Finally, the PowerPoint data set shows a range of 58% compressed size for $r = 1.00$ down to 37% for $r > 0$, indicating that superfingerprinting would not detect redundancy, which is still significant.

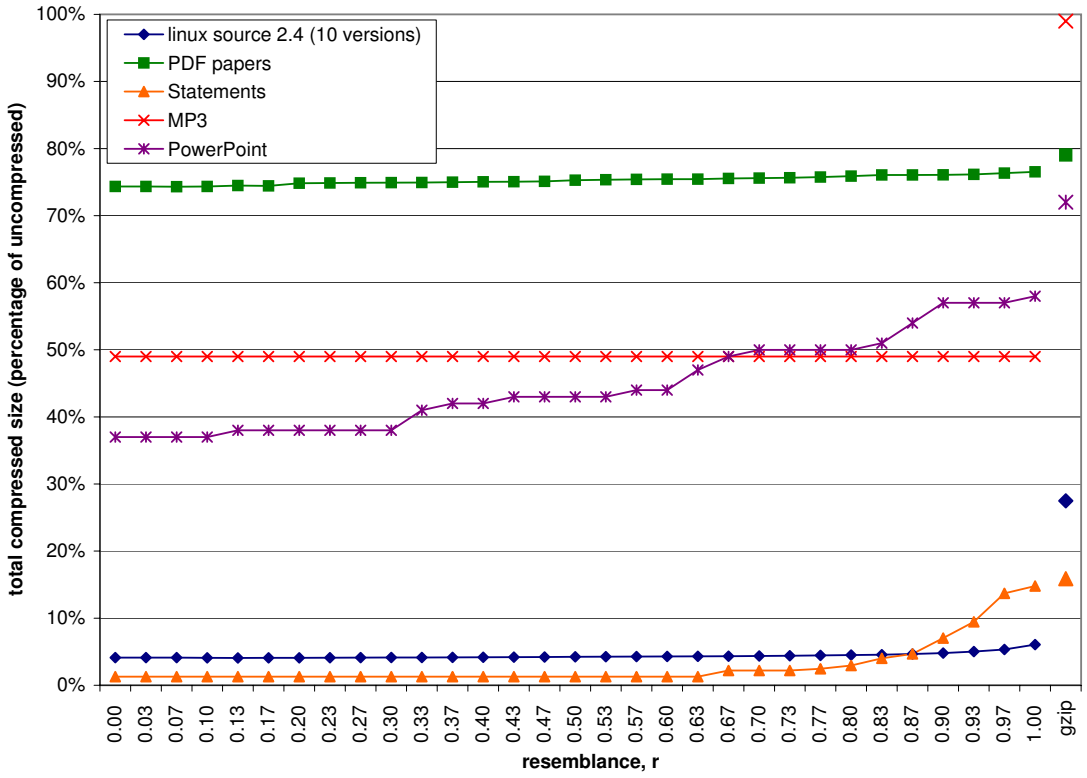


Figure 6.15: Storage used when delta applied: resemblance $(F1, F2) < r$ vs. gzip

6.4.4 Performance

In practice, space efficiency is not the only factor used to choose a compression technique; we briefly discuss some other important systems issues such as computation and I/O performance.

The chunking approach requires less computation than delta encoding. It requires two hashing operations per byte in the input file: one fingerprint calculation and one digest calculation. In contrast, delta encoding requires $s+1$ fingerprint calculations per byte, where s is the sketch size. It also requires calculating the deltas, even though this can be performed efficiently, in linear time with respect to the size of the inputs. Additional issues with delta

encoding include efficient file reconstruction and resemblance detection in large repositories.

To write chunks to the CAS, a hash table or distributed hash table must be first examined to determine placement of a chunk based on its chunk ID. Unlike traditional file systems, which can place object data based on directory, file name, or block number within a file, the chunk ID is intended to be globally addressable. During the file store operation, a write can be placed anywhere according to the mapping function provided by the CAS. However, during retrieval, chunks must be collected and retrieved. Over a distributed store, the random distribution of the chunk IDs lowers the probability that any single storage node contains all the data. However, the small sizes of chunks can make performance similar to a completely fragmented traditional file system. Wise engineering, such as clustering or ordering requests for a file, can improve performance.

The delta encoding method requires more compute resources than chunking and is made up of three main phases: computing a file sketch, determining which file is similar, and computing a delta encoding. Currently, the most costly phase is computing the file sketch due to the large number of fingerprints that are generated. For each byte in a file, one fingerprint is computed for the sliding window and another 30 fingerprints are computed for feature selection. The shingling performance in our prototype (CPQ hardware, Table 4.5) was approximately 1.6 MB per second, far less than the 30 MB per second disk read performance. We felt this performance was unacceptable, so through substantial coding optimizations, we were able to improve throughput to 11.4 MB per second, and later when running on the PL (Table 4.5) hardware, to 19.7 MB per second.

Fortunately, we believe that improvements in the fingerprinting implementation, the ability to parallelize the operation, and some heuristic modifications can easily increase the shingling performance by a factor of ten.

The second operation, locating similar files, is more difficult. Our prototype implementation was not scalable since it compares a new file against all existing files that have already been stored. Scalability in searching for a similar file is a particularly interesting area of our research. We are also optimistic that large-scale searches are possible given the existence of web-scale search engines that index the web using similar resemblance techniques [15].

The two techniques exhibit different I/O patterns. Chunks can be stored on the basis of their identifiers using a (potentially distributed) hash table. There is no need for maintaining placement metadata and hashing may work well in distributed environments. However, reconstructing files may involve random I/O. In contrast, delta-encoded objects are whole reference files or smaller delta files, which can be stored and accessed efficiently in a sequential manner, though the placement in a distributed infrastructure is more involved.

A couple additional issues exist for delta encoding that are not present with chunking. Because delta encodings imply dependency, a number of dependent files must first be reconstructed before a requested file can be retrieved. Limiting the number of revisions can bound the number of reconstructions at a potential reduction in space efficiency. Another concern that might be raised is the intermediate memory requirements; however, in-place reconstruction of delta files can be performed, minimizing transient resources [23]. At first glance, it would appear that the dependency chain and reconstruction performance of delta files might be lower than reconstruction of chunked files, but since reference and delta files are stored as a single file stream and chunking may require retrieval of scattered data—especially in a populated chunk CAS—it is unclear at this point which method would produce worse throughput.

6.4.5 Relationship Between Resemblance and Storage Efficiency

We validate our hypothesis that using inter-file compression between highly similar files is more efficient than compression between dissimilar files. Before engineering a system that uses significant computing or storage resources, we first show that the gains are significant, and that they are directly related to resemblance.

Our evaluation proceeds as follows. First, we manufacture a synthetic data set from two random starter files such and compute intermediates that are partially different between the two files. With this set of files, we run the *dcsf* algorithm and compute the amount of storage each file would take if it were to be compressed using delta compression against the most similar file already stored. We repeat the experiment with user data.

Figure 6.16 illustrates the direct relationship between delta compression on similar files. Using a synthetic data set of 201 files, we used *dcsf* to first detect similar files and then used *xdelta* to compute against the most similar file detected. The resulting delta file size on the vertical axis is plotted as a fraction of the size of selected reference file.

We created a synthetic data set using the algorithm and code used to evaluate the *zdelta* delta compression tool [162]. Random data was written into two starter files, f_0 and f_1, each of size 1,048,576 bytes (1 MB), to ensure zero resemblance. Next, intermediate files, also 1 MB in size were "morphed" by "blending" data from each of the two starter files using a simple Markov process parameterized by q and forming non-linear similarities. As stated in the *zdelta* technical report: "[The] process has two states, s_0, where we copy a character from f_0, and s_1, where we copy a character from f_1, and two parameters, p, the probability of staying in s_0, and q, the probability of staying in s_1." Within our experiments, we set $q = 0.5$ and vary p from 0 to 1 at 0.005 increments.

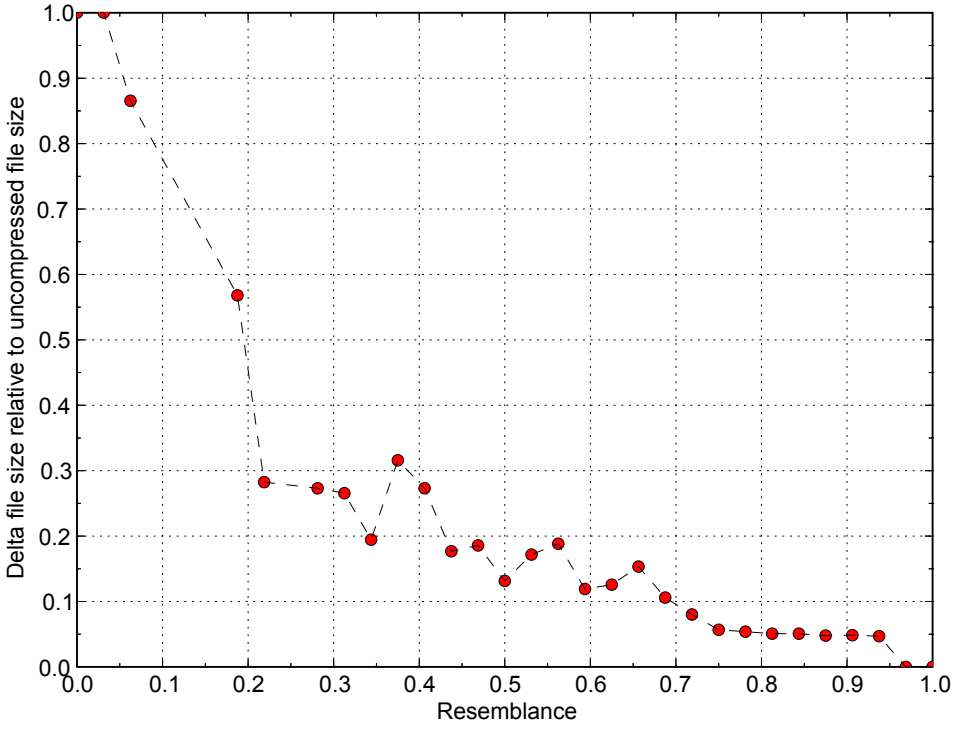


Figure 6.16: Relationship between resemblance and delta compression against uncompressed data

6.5 Discussion

Data compression aims to eliminate redundancy. In a system where many potential forms of redundancy might exist, applying the largest granularity of data compression will result in the largest efficiency gains. In addition, compression efficiency is strongly data dependent. It can be useful to think about the domain of data compression as applying to a spectrum. At one end, file data entropy is very high, and information theory will say it would be difficult to find any redundant data. Files of this type exist, either as random or seemingly random data, or as previously compressed data. For these files, virtually no similarity detection will help; however

redundancy can still exist when multiple copies of files exist. In this case, large-grained feature extraction results in small numbers of features that can be detected quickly and accurately. At the other end, files are stored that are slight variations of others; content such as human-readable text, computer-generated data, or modifications of previously stored data exist widely. In these cases, when data similarity exists, feature selection must be more fine grained.

Redundancy is highly data dependent, and so methods to eliminate redundancy will vary greatly. While the scope of data compression has typically worked within a limited context such as a file, data compression in storage has a much larger context to work within. Unfortunately, suppression of duplicate storage is not an optimal solution. Likewise, delta compression against single files is not either. The logical next step is to use the best solution possible when there is low cost. Because the act of compressing data, *i.e.* encoding it, is separate from modeling the compressed data, *i.e.* detecting the opportunity for compression, we aim to incorporate data compression in an opportunistic manner.

Modeling data compression can be difficult. When using heuristic methods, it's not always possible to produce results with the highest accuracy or with optimal results. Furthermore, our methods to detect similarity depend on a number of parameters that must be tuned, but it is not an practical matter to fix all parameters *a priori*. While we have shown that there is some tolerance to widely varying parameters, for instance sketch size, the size of the sliding window for shingling, and expected chunk sizes, there is no guarantee that these will work well for all data. The common result that good window sizes are within a range of tens of bytes is empirical in nature and may depend largely on a particular class of documents.

The reliance on some estimates may yield non-optimal results. In the case of sketches, a compression model relies on the resemblance, which is strictly an estimate for document similarity. The coverage of the selected feature data is a small fraction of the overall data set,

yet the expectation that surrounding data will be similar may not be a good assumption. Our experiments have shown that estimates for resemblance correspond proportionately to more efficient data storage, but these results are born from empirical data.

6.6 Summary

In this chapter we have described several ways to improve storage efficiency through data compression. Data compression, which can occur within a single file stream, or across files, eliminates redundancy.

The *whole-file hashing* method suppresses storage for identical files *i.e.* unity resemblance. The *chunk-based storage* method subdivides files and suppresses storage of individual chunks, with some overhead storing lists of identifiers. *Delta compression between similar files* computes sketches and compares them against sketches for stored files. Resemblance is a good indicator of the potential for file storage.

We developed a chunk compression program, *chc* to compress a single file into chunks by suppressing multiple storage of chunk data. The evaluation of this prototype provided results including chunk size distribution, and storage efficiency as a function of the distribution.

In DCSF, data compression rates vary as a function of many parameters including fingerprint window size, the delta chain lengths, and the degree of dependence. We evaluated several parameters and have shown that delta chains with length equal to zero or one are not as efficient as longer chains. We also evaluated the effect of delta chain length and degree of dependence on total space efficiency and have shown that differences in reference file candidate selection can dramatically change delta chain modeling, which may be useful for improved reliability models.

We compared chunk-based storage and delta compression storage. Delta compression

provides finer grain differences between similar data. These results pointed to the need for an integrated storage architecture that would allow both chunk and delta storage to be used to maximize the best of both methods.

Chapter 7

Storing Data

Once redundancy has been eliminated, data is ultimately stored in a unified content-addressable storage, or CAS, subsystem. In this chapter we describe the PRESIDIO Virtual CAS, or VCAS, which presents a unified storage interface externally but stores and reconstructs data using virtual content-addressable objects. First, we provide an overview of CAS systems and our contribution, the Virtual CAS. Second, we examine a number of properties in turn, namely the CAS objects, metadata, storage operations, reliability, and our prototype implementation. Third, we evaluate our VCAS design and prototype implementation. Finally, we summarize our design and results.

7.1 Overview

Content-Addressable Storage (CAS) provides many advantages for storing and accessing immutable data and recent industrial adoption shows that it is fast becoming a conventional means for recording archival data. Programs to compute a *content address* (CA), a *hash* or *digest* over a variable-length file, can do so at higher throughput than the bandwidth of

common magnetic disks. Cryptographic or one-way hash functions improve tamper-resistance because modifications to contents invalidate the address. Content addresses, which take up 16–20 bytes in our system, are smaller than filenames, and do not require directory organization to disambiguate names. For large volumes of data, the random distribution of content addresses allows storage system designers to create scalable addressing architectures so that file data is easily distributed across storage nodes or devices.

CAS systems also have a few shortcomings. Computing the content address is an operation that does not occur on block-based file systems; as a result, an entire file must be scanned before the address can be computed thus introducing a serialization dependency. Some CAS systems are designed with the assumption that low probabilities of collision of content addresses (hash values) are acceptable. Thus if a new file that is being stored hashes to an existing one, the storage operation is suppressed even though new content has not yet been recorded. The collision avoidance property of hash functions distributes addresses across the entire CA space, but by doing so, it eliminates any locality of name or reference, so CAS storage organization based purely on content addresses may cause more random access than linear access, thereby decreasing performance on disk-based systems.

Our contributions to CAS storage are the following. We extend the CAS model to combine virtual content addressing with a unified storage model and we also present a low-overhead VCAS storage implementation that provides further benefit by using temporal locality.

Figure 7.1 illustrates the relationship between the content-addressable storage sub-system and the efficient storage methods that depend on it.

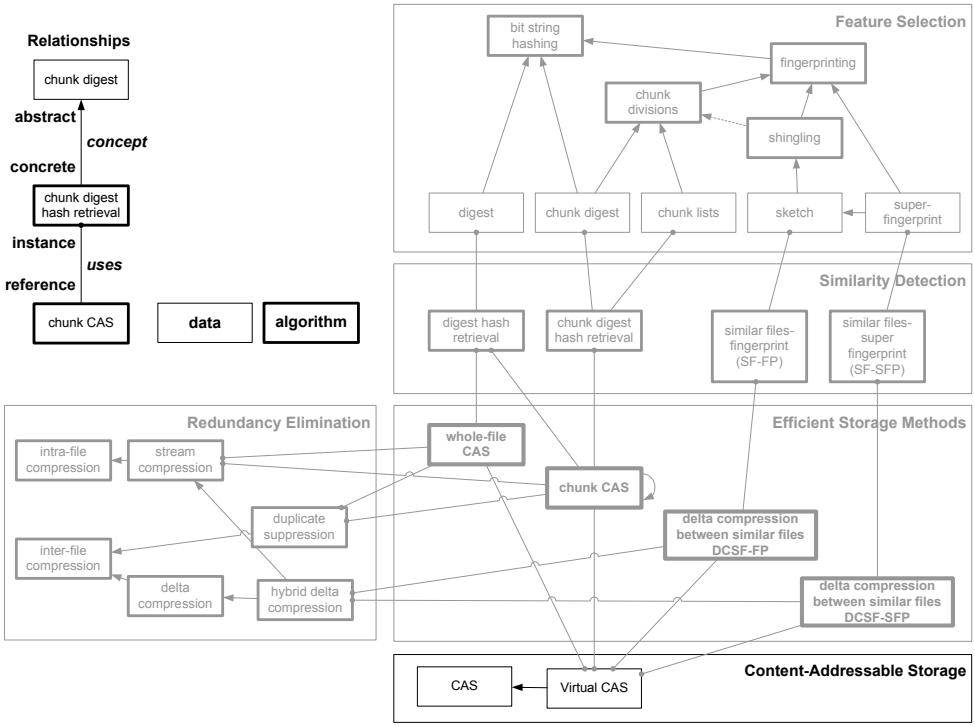


Figure 7.1: Content-addressable storage within the PRESIDIO framework

7.2 Content-Addressable Storage

We start with an abstract model of content-addressable storage. File content is identified by its content address (CA). A client program makes a *store* request to the CAS and is returned its CA. Later, another program sends a *retrieve* request with the CA and the original contents are returned.

We simplify the data storage model by separating file metadata from its contents. File metadata, which can include filename, ownership, location, and other, is serialized into a single stream. The metadata is also stored in the CAS and is identified by its CA. To improve the ease

of implementation, we use the same content address namespace for both file content and file metadata.

7.2.1 Addressing Objects

CAS systems share common addressing properties. Content stored in CAS are immutable. This permits the system to address by content instead of by location. Variable-length content can be reduced to a probabilistically unique identifier or address by hashing their contents using functions with very low collision rates. In many systems, one-way hashing functions are used in order to prevent intentional collisions of hash values.

Content addresses might refer directly or indirectly to a stored block of data. In other words, a hashing function like MD5 might compute its digest just over the block of data that is stored, or it might be computed over a handle that incorporates metadata that includes a content address within its structure that refers to raw content.

In order for a content address type to be reusable across a large storage corpus, its size, the function that computes values, and the bit representation of both the content as well as the address must be determined once and fixed for all time. Typical content address sizes start at 128 bits using cryptographic hash (digest) functions like MD5, which take a variable-length byte strings as input and produce a small fixed-length hash value.

Because addresses can only be computed once the entire contents of a file are known, the performance of the hashing function is a matter of practical importance. Although some fast hashing functions are known to produce low probability of collision with arbitrary input, they might not satisfy a system design requirement to prevent tampering of data by substituting stored contents with manufactured data that hash to the same address. In other words, cryptographic hashes are often a design requirement, but they can reduce performance, as we have

f	number of files
l_{file}	average file length, in bytes
l_{vcas}	average size of VCAS object, *e.g.* chunk
$c = l_{file}/l_{vcas}$	number of VCAS objects per file
$m = c \times f$	number of objects to be stored in the global VCAS
$n = 2^{ca_{bits}}$	number of addressable objects in the VCAS
$q = log_2(m)$	log_2 of the number of objects, m
$r = ca_{bits} = log_2(b)$	number of bits in the content address

Table 7.1: VCAS design parameters

seen in Table 4.4.

7.2.1.1 Hash Collisions

Although probabilistically infrequent, hash collisions can occur. We design our system to incur a low collision rate such that they are likely never to occur. We first set some design assumptions for the likely case as well as a conservative case, listing the main design parameters of interest in Table 7.1.

We start by designing a system to hold about 2^{30} or about 1.1 billion files. This is approximately equivalent to 2^{10} (or 1,024) storage nodes each with about 2^{20} (about 1.07 million) files each. We assume the average file size in a file system is less than 2^{14} bytes (16 KB) [154]. We have previously determined that a lower practical limit for files divided into chunks is at least 2^7, or 128 bytes. These parameters are listed in Table 7.2.

We represent content addresses as a fixed-length bit string, thus represent a space of $n = 2^{ca_{bits}}$ identifiable objects, where ca_{bits} is commonly 128–160 bits in CAS systems. However, for file objects with bit string length of $8l_{file}$, the actual space of unique objects is $2^{8l_{file}}$ and $ca_{bits} \ll 8l_{file}$. Although the universe of possible unique objects is large, we are interested in the probability of collision in the hash (content address) space.

	f	l_{file}	l_{vcas}	c	m	n	q	r	p	description
1	2^{30}	NA	NA	1	2^{30}	2^{128}	30	128	2^{-69}	MD5, 1 billion files, whole file objects
2	2^{40}	NA	NA	1	2^{40}	2^{128}	30	128	2^{-49}	MD5, 1 trillion files, whole file objects
3	2^{30}	2^{14}	2^{8}	2^{6}	2^{36}	2^{128}	36	128	2^{-57}	MD5, 1 billion files, 256 byte objects
4	2^{40}	2^{20}	2^{7}	2^{13}	2^{53}	2^{128}	53	128	2^{-23}	MD5, 1 trillion files, 128 byte objects
5	2^{40}	2^{20}	2^{7}	2^{13}	2^{53}	2^{160}	53	160	2^{-55}	SHA-1, 1 trillion files, 128 byte objects
6	2^{40}	2^{20}	2^{7}	2^{13}	2^{53}	2^{256}	53	256	2^{-151}	SHA-256, 1 trillion files, 128 byte objects

Table 7.2: Approximate collision probabilities, p, for specific VCAS design values

An object space of n objects does not have n^{-1} probability of collision; we consider the *birthday paradox* to evaluate collision instead. In other words, we compute p, the probability of collision as a function of m (the number of objects to be stored), n (the number of addressable objects) [65] with the formula:

$$p \approx 2^{2q-(r+1)} \tag{7.1}$$

whose derivation follows. The probability of success, *i.e.* no collisions occurring, is given by:

$$\begin{aligned} prob(success) &= e^{-(m(m-1))/2n} \\ &= e^{-2^q(2^q-1)/2(2^r)} \\ &\approx e^{-2^{2q-(r+1)}} \end{aligned}$$

Since for $1 - e^{-x} \approx x$ for small x, then the chance of failure is $p \approx 2^{2q-(r+1)}$.

Assuming uniform distribution of the hashing function, the collision rates are computed as probabilities of collision with a given set size. The different storage scenarios listed in Table 7.2 illustrate the effect on collision probability when the number of stored objects, content address size, and other related factors are varied.

Line 1 lists parameters one might find in a traditional whole-file CAS. We assume a requirement to store about one billion (2^{30}) files with using 128-bit CAs (like MD5) would yield a very low probability of $p \approx 2^{-69}$. In line 2, we increase the file count to one trillion (2^{40}) files and the collision rate across the entire CAS is extremely low, $p \approx 2^{-49} \approx 1.8 \times 10^{-15}$ that two files will collide.

In the next scenario (our target design) on line 3, we store about one billion files, each having average size of 16 KB, and an average sub-file object size of 256 files. These objects might represent small chunks or they could represent delta files. In either case, using 128-bit content addresses, we increase the number of objects stored in the CAS by a factor of l_{file}/l_{vcas}. We see that the probability of collision is still quite low, at 2^{-57}.

Line 4 represents a more conservative set of assumptions: one trillion files, 128 byte objects (not a very likely scenario due to the increased overhead of small objects), and a probability of collision of 2^{-23}. Lines 5 and 6 reveal how increasing the CA size to 160 bits (*e.g.* SHA-1) and 256 bits (SHA-256) further reduce the probability of collision significantly.

One last question remains: what probability of collision is satisfactory? A common design philosophy is to match or exceed the undetected disk bit error rate on magnetic storage which is typically 10^{-12} to 10^{-15} [131, 71, 62]. One might argue that this is a reasonable error rate, however there is a difference: in magnetic disk devices, undetectable errors occur at the block level. CAS storage systems like ours typically hash entire files and then identify the files by their CA. It would be more appropriate to evaluate sub-file hashing, for instance by first

subdividing the blocks. Comparing block-level hashing collision rates with undetectable block error rates is perhaps a more appropriate metric.

One criticism of CAS is its lack of strict correctness. Henson argues [60] that CAS systems that rely on hash equivalence for object equivalence is fundamentally the same as an error in design or implementation of hash tables in which a hash function is used to compute *hash keys*, but not comparing *values* when there is a collision in the keys. Systems like Centera compare the entire contents of files for correct operation at the expense of transmitting entire copies in all cases. One side benefit of CAS and network storage systems is that using hash identity suppresses transmission when data objects on both ends are hashed identically, even if their contents are not; network throughput rates can increase dramatically.

Another criticism of CAS is the use of an argument that compares of hash collision probabilities against the bit error rates [131] after device-based *error-correction codes* (ECC) are applied. Collisions are a metric of key collisions for CAS objects of arbitrary size and not the error rate for the recovery for a single bit. If the storage of a CAS object was suppressed due to a collision, then retrieval would likely be incorrect for a whole block whereas undetected ECC failure may only return slightly corrupted data. Furthermore, when we seek to eliminate redundancy by sharing common data through chunks or delta storage, the effect of a single error is magnified by the number of CAS objects (files) that depend on it. The failure modes between CAS and device are different, so comparing probabilities of error are difficult.

The study of CAS correctness is a topic of future work. Because of the exceptional instances of collision, we believe collision resolution can be handled in a manner by detecting collisions, definitively determining whether data is different, and then storing additional meta-data in CAS subsystems (or in our case, CAS storage *groups*) to indicate objects with colliding addresses.

7.3 Metadata

The PRESIDIO CAS stores metadata, adjunct information related to file contents. We distinguish *file metadata*, primitive file system-level information about the raw content of the file, from *rich metadata*, application-level information. While file metadata, like the size of file and modification date, are ubiquitous and can be specified easily, rich meta information like keywords, indexes, thumbnails, or other representations of a file, are specific to applications.

Our storage system is designed to store the most primitive metadata necessary. Because the mechanism to store any kind of file data—including metadata—is designed for low overhead and low redundancy, applications and storage layers built on top of the PRESIDIO VCAS can create their own metadata formats.

7.3.1 Archival metadata formats

File archives contain both file content as well as file metadata. We examined some Unix file formats for their size overhead in order to assemble a simple metadata format that could be used to archive and restore commonly used files.

Two Unix uncompressed archiving programs *ar* and *tar*, and two compression programs, *gzip* and *xdelta*, are common examples of programs that retain file metadata. We measured files with one byte length and then disassembled the file format to determine what would constitute a minimal set of file attributes.

In 7.3, we list two files, **a** and **b** to show the relative file overhead for each single-byte file. Because the *ar* file format was small, we extracted the fields that would be commonly used.

```
/usr/include/ar.h: (FreeBSD 4.8)

struct ar_hdr {
    char ar_name[16];       /* name */
    char ar_date[12];       /* modification time */
```

filename	size (bytes)	command to create file
a	1	`echo -n a > a`
b	1	`echo -n b > b`
a.a	70	`ar -q a.a a`
b.a	70	`ar -q b.a b`
a.a.gz	69	`gzip a.a`
b.a.gz	69	`gzip b.a`
a.tar	10,240	`tar -cf a.tar a`
a.tar.gz	122	`tar -zcf a.tar.gz a`
a.tar.gz	128	`tar -cf a.tar a; gzip a.tar`
ab.tar.gz	143	`tar -zcf ab.tar.gz a b`
b.delta	209	`xdelta delta a.a b.a b.delta`

Table 7.3: Archive file metadata sizes

```
    char ar_uid[6];         /* user id */
    char ar_gid[6];         /* group id */
    char ar_mode[8];        /* octal file permissions */
    char ar_size[10];       /* size in bytes */
#define ARFMAG  "`\n"
    char ar_fmag[2];        /* consistency check */
};
```

Next, we converted each field into a corresponding XML entity using a representation in text, thus producing a suitable replacement that is easily interpreted:

```
<?xml version="1.0"?>
<file>
 <name>presentations.tar</name>
 <ca>9797f2de010e6244cc39081ceeb0d447</ca>
 <mode>00740</mode>
 <uid>1005</uid>
 <gid>513</gid>
 <size>15083520</size>
 <mtime>1091490486</mtime>
</file>
```

Metadata was separated from file content and then converted into a text representation that was then stored as another content-addressable object. Figure 7.2 illustrates the storage of content and metadata and the data dependency.

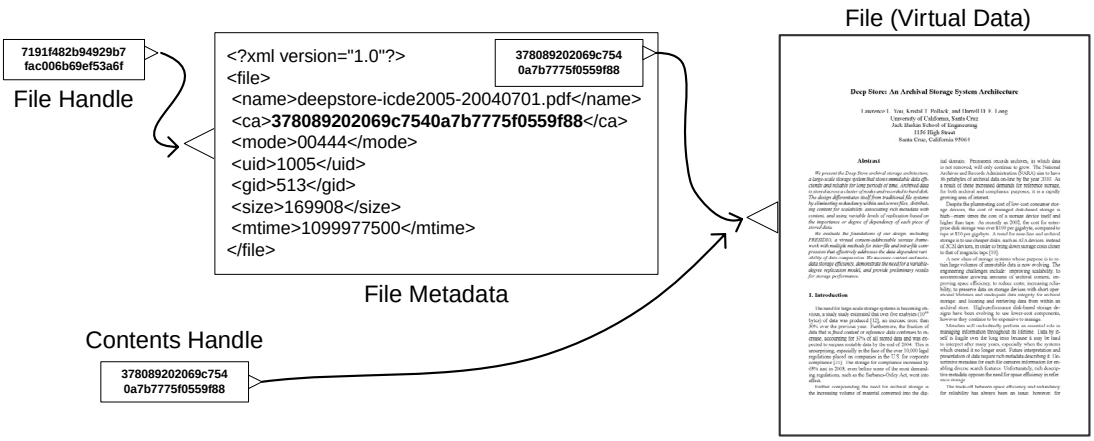


Figure 7.2: Metadata and file content

7.3.2 Rich metadata

Metadata will undoubtedly play an essential role in managing information throughout its lifetime. Data by itself is fragile over the long term because it may be hard to interpret after many years, especially when the systems that created it no longer exist. Future interpretation and presentation of data require rich metadata describing it. Descriptive metadata for each file captures information for enabling diverse search capabilities [38, 98]. Unfortunately, storing the rich metadata substantially increases file size overhead, when space efficiency is an important goal in reference storage.

Our work has shown that rich metadata storage may have different storage requirements than the archival data it references [174]. The first difference is that the metadata might not have the same immutability requirement. This is because features such as keywords may be extracted from the file based on changing criteria. Another reason is that file security permissions may change even though the file's contents remain immutable. The second main difference is due to the structure of metadata content. When using a representation such as XML,

increases in size due to metadata versioning can cause a linear increase in total file storage size when delta compression is used, while other content-specific compression mechanisms, like XMill for XML [86], have much higher space efficiency [16].

7.4 Storage Operations

Storage operations are implemented in the CAS program, a Unix library written in C++, and by an external interface written in C++ and Perl. The storage operations are declared abstractly with a modest set of functions, the most basic being the following two operations:

- Store object
- Retrieve object

The **Store object** operation takes as input a whole object, which is often a file, and returns a content address (CA). The **Retrieve object** operation takes a CA as input and returns the original file.

Locality of reference is not guaranteed. Content addresses are hash values with the goal of minimizing collisions, so the address by itself indicates no relationship to other stored data. On the other hand, the VCAS interface does not preclude the implementation from inferring locality of reference through caching or temporal relationships, *e.g.* deriving a closeness from the time files are stored into the archive.

Operations might be idempotent. Multiple instances of similar or identical files might not be stored multiple times. This is important so that applications that create duplicates or backup copies do not assume that storing more copies improves reliability.

Neither VCAS nor Deep Store nodes interpret stored content. Document preservation and interpretation of digital data is another difficult problem that is approached from a related,

but separate area of research of digital preservation. Our system stores the data that enable preservation strategies to be implemented more easily.

7.4.1 C++ Interface

A fragment of the C++ class declaration for the implementation follows. Most notable are the parameters for the first `Store` member function: the address is returned once the object is stored. Likewise, the `Retrieve` member function takes a single parameter for the address of the object and returns the data in buffers specified by output variables, `outData` and `outLength`.

For convenience, our object-oriented C++ implementation used polymorphic behavior to provide different "back-end" implementations. Because the `TCAS` interface is abstract, we were able to experiment with the underlying CAS by first implementing a file-system based solution and then later using a Berkeley DB implementation by overriding the factory method `CreateCASStore`. Overloaded functions provided additional benefit by allowing us to write partial blocks with a given address, necessary to handle large block writes.

```
class TCAS {
public:
                        TCAS();
    ...
protected:
  virtual TCASStore*  CreateCASStore() = 0; // factory method

public:
  virtual TContentAddress
                        Store(const void* data, size_t length);

  virtual void          Store(const TContentAddress& ca,
                              const TContentType& ct,
                              const void* data, size_t length,
                              size_t offset);

  virtual void          Retrieve(const TContentAddress& ca,
                                 TContentType& ct,
```

```
                                void* outData,
                                size_t& outLength);
  virtual void        Retrieve(const TContentAddress& ca,
                                TContentType& ct,
                                void* outData,
                                size_t& outLength,
                                size_t offset);
};
```

A C++ function that writes a file of size `casdatasize` bytes starting at the pointer `casdata` and a content type `ct` would make a call and return a content address object, `ca`:

```
static TContentAddress CASStore(TCAS& cas, const TContentType& ct,
                      const void* casdata, size_t casdatasize)
{
  try {
    TContentAddress ca = cas.Store(ct, casdata, casdatasize);
  } catch(TCASException& exc) {
    printf("CASStore Exception occured: %s\n", exc.GetString());
    throw;
  } catch(...) {
    printf("CASStore Exception\n");
    throw;
  }
  return ca;
}
```

The `WriteObject` member function implements the low-level non-virtual CAS function:

```
void
TCASDBStore::WriteObject(TContentAddress ca, TContentType ct,
         const void* data, size_t length, size_t offset)
```

The algorithm to write a new object into the underlying storage is straightforward:

Two other operations that would complete the set are **Delete object** and **Verify object**. These were not implemented. Deletion of VCAS objects might not cause data to be deleted from the store if files are still referenced. Hence, referenced file handles must be stored separately

Algorithm 1: Write data to CAS

```
this ← content-addressable-store
ca ← content-address
ct ← content-type
data ← pointer-to-data
length ← length-of-data
offset ← offset-from-start
(ca-header, ca-exists) ← this.ObjectExists(ca)
if ca-exists then
    update the ca-header reference count
else
    recno ← index-database.GetLastRecno() + 1
    (key, value) ← (ca, recno)
    index-database.Put(key, value)     write out megablock
    megablock-record-id ← cas-database.Append()
    (key, value) ← (recno, megablock-record-id)
    block-database.Put(key, header)
end if
```

and the file content (CAS objects) must be reference counted. The verification operation, had it been implemented, would test objects by first reconstructing them and then checking them against their content addresses.

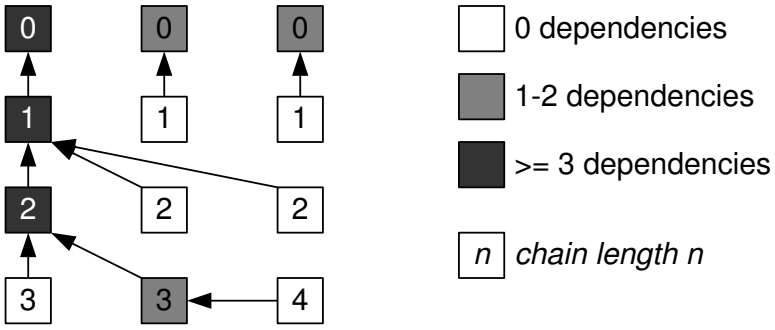


Figure 7.3: Delta dependency graph

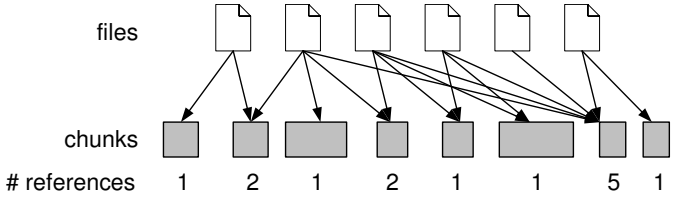


Figure 7.4: Chunk dependency graph

7.5 Reliability

We briefly discuss the effect of data dependency on reliability. The problem of reliability becomes much more interesting when data is stored in a compressed format. How do we reintroduce redundancy (for reliability) after we have worked so hard to remove it? Now that files share data due to inter-file compression, a small device failure may result in a disproportionately large data loss if a heavily shared piece of data is on the failed component. This makes some pieces of data inherently more valuable than others. In the case of using delta compression for stored files, a file may be the reference file for a number of files, and in turn those may be a reference file for another set of files.

We measured data dependencies in both delta and chunking experiments. Using a common data set of highly similar data, we ran *dcsf* (delta compression) and *chc* (chunk com-

pression) to determine the extent to which dependencies were introduced. With delta chains, a single file may quickly become the root of a large dependency graph, as shown in Figure 7.3. If a file is lost, the integrity of all dependent files is at risk. Figure 7.5 shows the chain lengths for a set of highly similar data (ten versions of Linux kernel sources) stored using PRESIDIO with only delta compression. The average chain length for the set of files was 5.83, but more interestingly, there were only five root reference files (chain length zero) for this file set. In other words, in the set of 88,323 files, all of the files depended on five files.

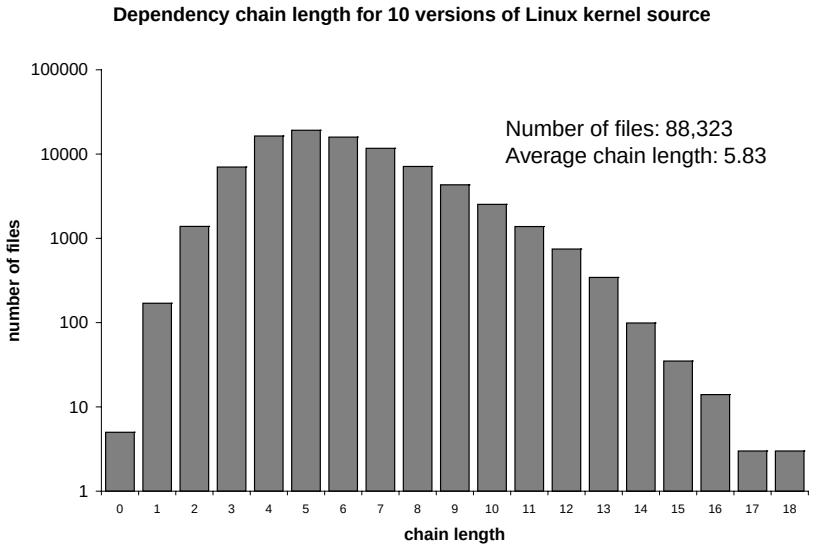


Figure 7.5: Delta chain length

Sub-file chunking also introduces inter-file dependencies, as depicted in Figure 7.4. If a large set of files all contained the same chunk, for example a regularly occurring sequence in a set of log files, the loss of this small chunk would result in the loss of a large set of files. Figure 7.6 shows the number of files dependent on shared chunks for the same set of files used in Figure 7.5, stored using PRESIDIO with only chunk-based compression. The large peak for chunks with one reference shows the number of unique chunks, and the peaks seen at 10, 20, 30

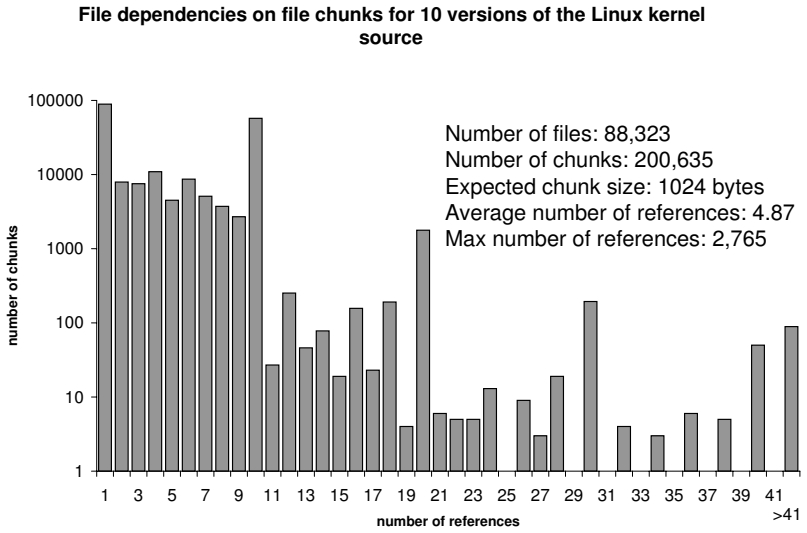


Figure 7.6: Chunk dependencies

and 40 occur since the files are 10 versions of the Linux kernel source. Of the 200,635 chunks stored, only 89,181 were unique, with an average of five files referencing a chunk. The data extends out to 2,765 references to a single chunk; however, it was truncated to 41 for the graph in the interest of space. There were 72 chunks not shown with more than 41 references, 17 of which were over 100.

A new reliability model for chunk- or delta-based compression would place increased value of a stored object based on the uncompressed data, and not a unit value for each compressed value. So we would consider the five reference files in our first example and our chunk with 2,765 references from our second example to be more valuable than less referenced data. To protect the more valuable data we would like to store it with a higher level of redundancy than less valuable data in order to preserve space efficiency while minimizing the risk of a catastrophic failure.

Another issue that must be considered for preventing catastrophic failure is that of

data distribution. Consider files and/or chunks distributed randomly across devices in a storage system. If a device in the system were lost due to failure, the effect on the files stored would be devastating. Depending on the number of devices and the degree of interdependence of the data, it would be likely that a file in the system would have a chunk of data lost, or a missing file in its delta chain, preventing future reconstruction.

Storage systems that distribute data for security or for reliability exist but they assume data has unit cost. Distributed storage systems such as Petal [84], OceanStore [82], Interarchival Memory [54, 35], FARSITE [3], Pangaea [138, 139], and GFS [51] address the problem of recovering data from a subset of all stored copies. In these systems, both plaintext and encrypted data are distributed, and recorded with simple replication or error-correcting coding schemes. However, they do not distinguish between degrees of dependent data.

It is clear that reliability guarantees for a system storing data with inter-file dependencies is a particularly difficult problem. Building a model that derives the value of data and developing a strategy for data placement are the subject of future work.

7.6 Implementation

We implemented a CAS server that operates on a single host. It is a basic design aimed at storing a large number of variable-length data blocks. Primary considerations were to index each block by its content address, to minimize storage overhead per block, to present a small set of functions, and to improve performance with easily implemented modifications.

We start with a summary of our block storage strategy. First, the CAS divides an input file into create smaller pieces of data over which to identify duplicate or similar data. This data may potentially be shared with other files, so re-write of previously written data is suppressed. Second, the CAS data are written in order to storage segments, called *megablocks*.

Third, the megablocks are indexed to internally address the VCAS storage. Fourth, the indexed megablocks form *groups*, which can be distributed across the storage system. The implementation uses these principles to record variable length block data using flat files and databases storing key-value pairs.

7.6.1 Design Overview

Reducing storage overhead in an immutable content-addressable store is a simpler problem than in read-write file systems. In a fixed-content CAS, objects are written but never modified; this immutability property allows a system to organize data in a way such that the written data does not need to be moved or resized to accommodate append or truncate operations. In contrast, read-write file systems need to allocate extra space when files are appended and most designs use fixed-sized blocks, which exhibit *internal fragmentation* when not completely full. Furthermore, file system performance goals typically require *extents* or other sequential grouping strategies to better utilize the highly beneficial sequential throughput that hard disks exhibit as opposed to high latency when random block placement is used.

Like many storage systems, our design assumptions were aimed at balancing space efficiency against read and write performance. To reduce per-file and per-block overhead that are commonly found in storage systems using fixed block sizes, we used variable-length blocks. To facilitate high write throughput, we used a lazy write placement model by writing sequential data. To help reduce read latency, we indexed our data by using two levels of indexing, the first containing primary indexes to record offsets similar to the *indexed-sequential files* [169] or *indexed sequential access method* (ISAM), and second, to use a hash table to map content addresses to the primary indexes.

7.6.1.1 Sequential Block Storage

Another problem is to improve storage read and write performance by exploiting locality of reference. In file systems, files or directories may sometimes be placed together actively or as a matter of the write patterns. In contrast, content-addressable storage is designed to use addresses with random distribution of values, making locality of reference by address impossible.

A common theme in file system design is to improve sequential block storage layout by grouping blocks or files together such that related data are accessed quickly. File system actions exhibit many behaviors. For instance, files can be related in their address (namespace), or they can be related by time. Fortunately, file access patterns are typically related by time and ordering, making it possible to improve performance by using temporal locality.

The Log-structured File System (LFS) [134] opportunistically writes blocks linearly onto a storage device as they arrive, having the effect of grouping temporally related blocks together. Results from HighLight, a *hierarchical storage management* (HSM) system extending LFS, suggest that by using a migration policy to group by access time instead of namespace, the storage system can achieve high bandwidth and low latency by using secondary caching with tertiary storage [78].

7.6.1.2 Megablocks: Sequential Block Storage

Megablocks are user-level container files containing temporally-grouped variable-length data. The underlying block storage implementation is similar to *chunks* used in the Google File System (GFS) [51], large flat files for containing parts of other files. In our implementation, the large container files are also flat files with no special metadata attributes and range in size from 16 MB (represented by 24 bits of offset) or to 4 GB (represented by 32 bits

of offset). We selected this range for the following reasons. Small numbers of large file sizes require less overhead per file than larger numbers of small files. Clustered file systems like GFS and GPFS [142] improve access by multiple clients when using larger block allocation sizes larger than 64 KB (versus file system allocations typically at 4 KB) due to the lower per-block overhead for managing the block storage and associated file allocation tables across multiple storage nodes. Very large files would not be limited to these boundaries because of the nature of the VCAS storage object model, which supports arbitrarily large files through composition. In other words, a very large original file could be stored and reconstructed from concatenated blocks.

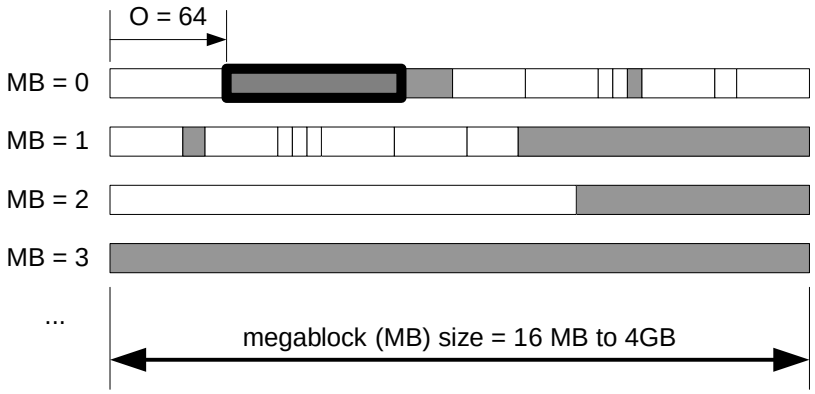


Figure 7.7: Megablock (MB) storage

In Figure 7.7, megablocks zero (MB = 0) through three (MB = 3) are shown as horizontal rectangles representing flat files. File content is shown as dark gray rectangles, which represent single VCAS objects. The object with the heavy outline is located with an offset of 64 bytes from the beginning of the file (O = 64).

Objects are placed within megablocks sequentially to reduce storage overhead from unused portions of blocks and to maximize contiguous writes. Stored data (white boxes) are

stored contiguously. Unused data (dark gray boxes) may be present. Megablock fragmentation can be temporary; a periodic *cleaner* operation (not yet implemented) will reclaim unused space. A fixed megablock size from 16 MB to 4 GB is selected for uniformity across nodes and the ability of group migration. Compared to file systems, which typically have block sizes in kilobytes, this range of megablock sizes is better matched for large-file storage on cluster file systems such as GPFS and GFS. Files larger than the size of a megablock are divided into chunks smaller than or equal to the size of a megablock and stored as virtually, as a Concatenated Data Block.

7.6.1.3 Group Storage

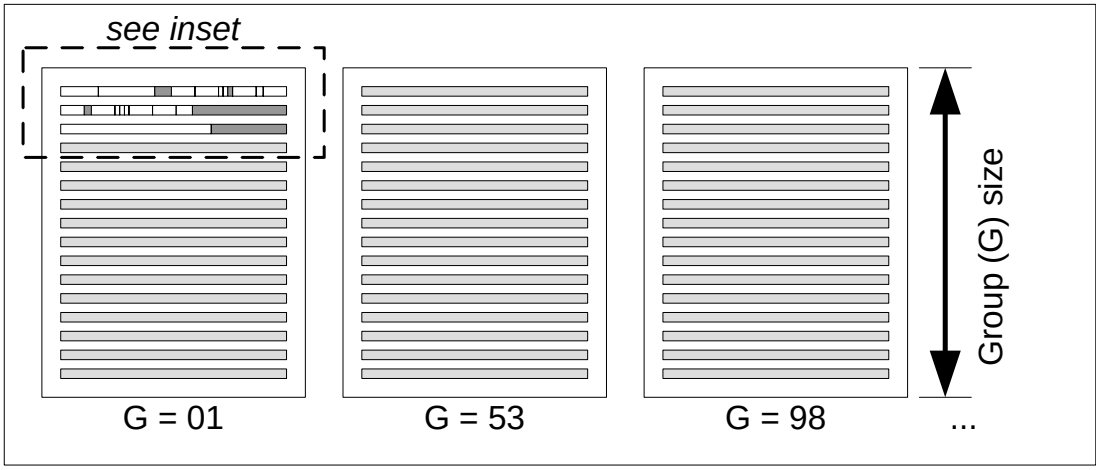


Figure 7.8: Group (G) and node storage

A storage *group*, shown in Figure 7.8, contains a number of megablocks, placed on a recording device using a storage mechanism such as a cluster file system. In this example, the group numbers are 01, 53, and 98. Each group is stored on the node of a server cluster. For reliability, groups can be recorded with varying levels of replication or coding. A small

distributed hash table is maintained on each Deep Store node to allow a single node to look-up a node number from the group number. Groups can be migrated from existing nodes to newly added nodes to distribute load. The simple group and megablock structure is easily ported to new large-scale storage systems, and allows group migration to yield efficient storage for a wide distribution of file sizes, including small objects such as file metadata, with very small object-naming overhead. The group also serves to contain a naming space that is unique across a cluster.

7.6.1.4 VCAS and Object Types

We introduce the PRESIDIO virtual content-addressable store (VCAS). The main data types in PRESIDIO are illustrated in Figure 7.9.

Handle: a file handle that contains a *content address* (CA), such as an MD5 or SHA-1 hash. Our prototype stores only an MD5 hash (16 bytes), but we anticipate that we will augment the handle with a small amount of metadata to resolve collisions.

Constant Data Block: a content-addressable data block of variable size containing a string of bits that is stored literally (*i.e.* raw binary data).

Virtual Data Block: a content-addressable data block. Each block contains a type code such as "constant" (signified by K), "concatenation" or "chunk list" (Σ), and "differential" or "delta," Δ). Each block has a content address; the contents are reconstructed polymorphically but stored virtually. Handles can be embedded within other blocks of data. The polymorphic behavior is flexible because it allows a single address to map transparently to multiple instances or alternate representations.

Objects are stored by content; each lettered box indicates the content address type: **C** for "chunk," **R** for "reference file" (virtual or real object), and **D** for a "delta file" (also virtual

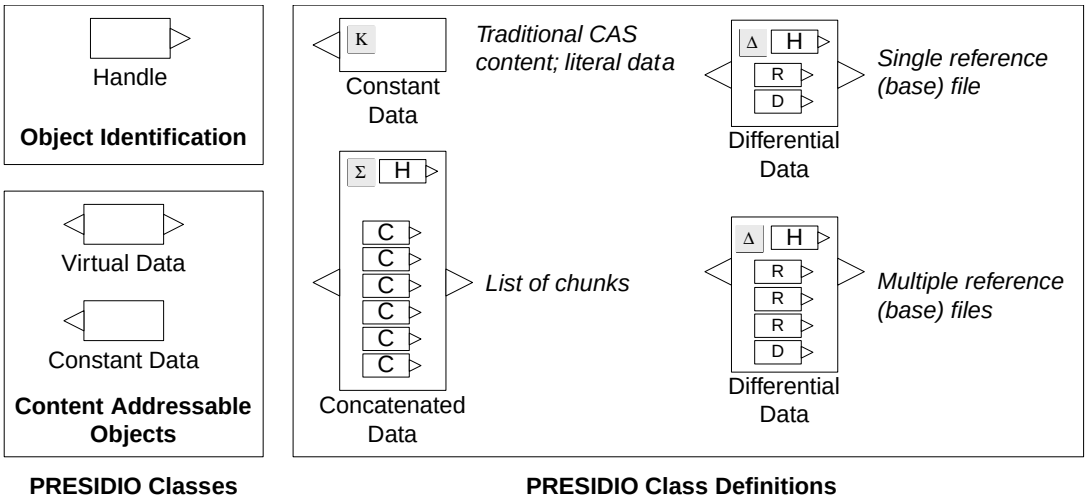


Figure 7.9: PRESIDIO data classes

or real). Embedded handles, **H**, contain the hash for the whole file.

Our simplified content-addressable storage uses simple storage objects and addresses to those objects.

CAS objects are untyped data made up of a sequence of bytes. These objects are used to persistently store a wide range of data including whole files, partial files, delta files, chunks, sketch data, metadata, and so on. Because our CAS uses only one type of storage object, storage operations are specified and implemented easily.

To locate a single content-addressable object, first a handle is presented to the Virtual Object Table. The handle's content address is used as a hash key to look up the storage location of the Virtual Data Block that is referenced by the table. The Virtual Data Block is retrieved and the handle is compared for its identity.

Reconstruction of the real image of the data block follows. The framework currently allows for three different types of reconstructible data, or any combination of them.

Constant Data (*K*) If the Virtual Data Block is a Constant Data Block, then the reconstruction is complete and the data block is returned. A variation of this is the Compressed Constant Data Block (*Z* type, not shown), which is compressed using the *zlib* stream compression library.

Concatenated Data (Σ) If the Virtual Data Block is a Concatenated Data Block, reconstruction consists of iterating through the list of handles and retrieving them recursively. The concatenated data is returned serially to the storage service interface. This type is used to store and reconstruct files from chunks, but not strictly limited to chunk storage. For example, very large files can be divided into smaller files and represented as Concatenated Data.

Differential Data (Δ) If the Virtual Data Block is a Differential Data Block, then reconstruction consists of first reconstructing the Reference Block (R) and the Delta Block (D). A delta reconstruction, using a delta-based compression program such as *xdelta*, applies the delta file to compute the resulting Version Object which is returned to the storage service interface. Note that Version Objects can refer to multiple Reference Blocks.

The framework can be extended to different types of Virtual Data Blocks; for instance, a single version (instance) of a file's metadata can be extracted from a data block storing the entire version history for that file.

Figure 7.10 illustrates the simple relationship between constant data CAS objects and virtual data CAS objects. Object data (files and internal metadata) are stored as small objects. A single handle can be used to store object data. Each entry in the Virtual Object Table consists of a *group* number (**G**), a *megablock* number (**MB**), and an *offset* in the megablock (**O**). (Additional information, such as the block type and length, are not shown.) Our prototype, which uses 16 bit group and megablock identifiers and a 32 bit offset, addresses a maximum of 16 exabytes (18×10^{19} bytes).

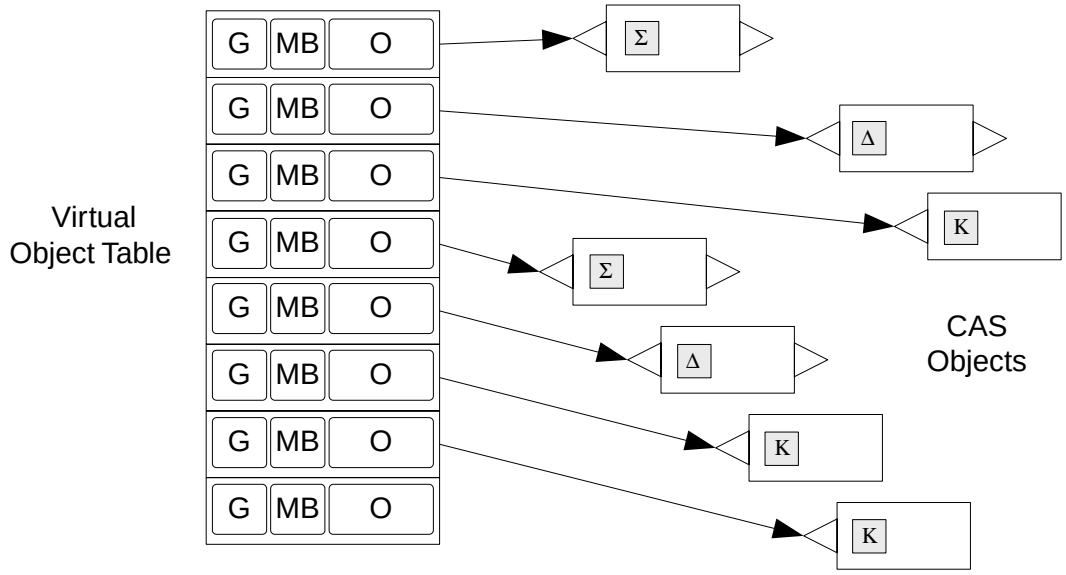


Figure 7.10: PRESIDIO content-addressable object storage

7.6.2 Indexing

Once data is recorded, we provide an index to map content addresses to megablock positions. Our implementation uses two level indexing: a Berkeley DB *Recno* database [149] (record number as keys) that maps a sequential index to a variable-length block (record) location and then a simple Berkeley DB *Hash* (B-tree hash table) database that maps the CA to the sequential index. These two levels of indirection achieve two goals: to allow sequential data to be written quickly, and for CA retrieval to be fast by using very small hash entries consisting of CA and *recno*.

7.6.2.1 Single-Level Hash Table

We started with the most straightforward implementation, a Berkeley DB *Hash* table of *key* and *value* pairs. The content address (CA) was the key and the content or data block was

the value.

Although the *Hash* implementation was simple, we immediately discovered several problems. The first, and most important, was that Berkeley DB stored the combined keys and values pairs in memory. With expected block sizes in the range of several kilobytes, the ratio of the size of value to the size of key would be very high. This caused the number of keys stored in memory to be low.

The second problem we discovered was that random access read and write patterns did impose a large performance penalty. Content addresses are implemented as hash values computed from randomized functions; because of their random distribution, we used the content addresses as the Berkeley DB key. During both file store and retrieve operations, blocks and their CAs were written to and read from the Berkeley DB CAS. During writes, after the large hash tables exceeded available RAM, the database began to thrash and performance suffered greatly. Similarly, during reads, random access retrieval (due to the CAs with random distribution) severely degraded database performance due to no locality of reference.

The simplest implementation was not going to provide acceptable performance or scalability so we developed an approach to construct an additional level of indirection next.

7.6.2.2 Indirect Table

In the second iteration of the design, we examined the likely behavioral pattern of the data that was to be stored and retrieved from the system by observing the results of the first implementation and the expected storage operation usage. The first observation was that Berkeley DB random access read operations should avoid touching disk if at all possible. The second observation was that writes should be written as they arrive.

Berkeley DB *Hash* storage method gives the best random access performance if all of

the keys and values are in memory [144]. The reason is that the implementation, based on *linear virtual hashing* (*linear hashing* or LH) will spill hash table buckets to disk when the memory buffer limit is exhausted. Therefore during a database lookup, if the hash subtable is not in memory, it must first be read from disk. When all accesses are random, *i.e.* all content addresses have random distribution, and the set of content addresses accessed grow, then eventually a single Berkeley DB hash table will grow beyond available real memory. The solution is to maximize the number of entries that can be stored by minimizing the combined key-value pair size. The obvious design change was simply to store the content address as the key and a secondary index (a 32- or 64-bit ID) to a separate storage database.

Next, block contents were organized by writes. We surmised that write performance would be highest if data was written sequentially in the order it was received. Like log-based file systems (LFS), write performance can be enhanced if data is appended sequentially instead of first taking the step of allocating data first and then writing it next. In our simple block storage scheme, we appended blocks as they arrived into a megablock file; when each megablock file reached a limit, it was closed and a new megablock was started.

7.6.2.3 Comparing *Hash* and *Recno* Writes

We compared the performance and storage overhead of the two types of Berkeley DB 4.2 database, *Hash* and *Recno*. Using the ProLiant ("PL") hardware listed in Table 4.5, we wrote 10 million entries of size 96 bytes (16 bytes for key and 80 bytes for value).

The cache size was set to 256 MB on a 2,048 MB RAM machine. Each record was written sequentially using an MD5 hashed key (not used in Recno). Once the database was closed, the resulting `cas.db` file size was measured. Notably, the hash performance of random access was hampered significantly as the 2.39 GB file exceeded both cache size and total RAM,

Hash	Recno	
10,000,000	10,000,000	entries
16	16	bytes/key
80	80	bytes/value
24,735.78	103.167	seconds
0.002473578	0.00001032	seconds per entry
2.473578	0.0103167	milliseconds per entry
cas.db	cas.db	database size
2,574,974,976	892,649,472	bytes
96	96	key+value size
257.5	89.3	average record size
161.5	-6.7	overhead per record (bytes)
168%	-7%	overhead per record (%)

Table 7.4: Berkeley DB *Hash* versus *Recno* writing 10 million entries

key	value
content address (CA)	block-recno

Table 7.5: PRESIDIO CAS Content address (CA) *Hash* index

causing disk thrashing. The results of this experiment show how it was necessary to change to design to an indirect table.

In summary, we introduced an indirect table consisting of a mapping from CA to recno (using the Berkeley DB *Hash* method) and then a separate mapping from recno to a megablock record (using method *Recno*). These correspond to the key-value pairs shown in Tables 7.5 and 7.6, respectively.

We also took the opportunity to add a very small amount of easily-interpreted meta-data into the (*Recno*) block table, as seen in 7.6. This achieved two things: the first was to reduce

key	value
block-recno	(content type, refcount), (group-ID, megablock-ID, offset, length)

Table 7.6: PRESIDIO CAS Block index *Recno* table

the number of operations to access the megablock's *content type* (class), *reference count*, *group ID*, *megablock ID*, *offset* (offset from the start of the megablock file to the block), and *block length*. The second was to restrict megablock files only contain block data. Note that Table 7.6 is an implementation example of the abstract content-addressable object storage scheme illustrated in Figure 7.10. A key consisting of *block-recno*, is a recno value for the block, and the value is a combined mutable tuple of content type and refcount with the *group-ID*, *megablock-ID*, *block offset* into the megablock file, and *block length*.

7.6.2.4 Content Types

The *content type* is one of the following: **Raw Content**, raw, uncompressed data; **ZLibData**, zlib-compressed data; **Concatenated Content** a list of CA; **XDelta1 Content** a file reconstructed from a reference file and a delta file using *xdelta 1.1.3*; and **Metadata**, CAS metadata. These are the example set from which we can reconstruct content and metadata in our initial implementation.

7.6.2.5 Content Addresses

The content address type is a binary scalar, 128 bits or larger in size. For convenience and because it is a well-specified hashing function, we use the MD5 algorithm.

7.7 Discussion

Although our problem generally solves a file storage problem, there is a need to establish a foundation for storing data. Content-addressable storage systems present a different interface than traditional file systems: instead of using location and meta naming as the means to identifying a file, its name is derived from the file content. From this idea we expanded the

idea that the name, or rather a content address, is more important than the representation of the object itself.

Once we determined naming was the first class object of the system, we modified the idea that storage efficiency should be dependent on data representation such that a client of the system should only be concerned with the naming and its virtual representation; furthermore, file compression methods should be able to represent their large-grained objects within a single storage architecture. By creating the VCAS, all needs could be met.

Although our initial prototype is sufficient for investigation and experimentation, we believe there is room for improvement. What traditional file systems do better than our system is to exploit locality of reference, whether it be from storing files in directories together, sharing i-nodes, using block extents, or using temporally-related or access-dependent caching or prefetching. In contrast, our goal for improving storage space efficiency improves *locality of associativity*, which does not clearly align with high performance. As such, it will be important for future work to incorporate inherent locality even when the content addressable architecture does not directly reveal such relationships.

7.8 Summary

In this chapter we have described content addressable storage (CAS) and virtual CAS (VCAS) systems and their properties, including the use of hashing for addressing, the impact of hash collisions on CAS storage design, file and object metadata and their overhead. The VCAS architecture is an underlying storage system piece which can be used onto which redundancy elimination storage methods including chunk-based storage, delta-based storage or hybrids can be implemented.

We have described design and implementation of a sequential large-block storage

facility into which we can store large numbers of VCAS objects with low overhead and with high write throughput by using a multilevel indirect storage mechanism employing both record-based and hash-based tables.

The overview of the PRESIDIO implementation includes the programming interface and operations for storage and retrieval and implementation notes for developing a high-performance content-addressable object store.

Finally, we have identified new problems in archival storage reliability due to the the increased dependence due to the elimination of redundant data.

Chapter 8

Progressive Compression

The Progressive Redundancy Elimination of Similar and Identical Data In Objects framework, or PRESIDIO, is a combination of an algorithm and framework to compress data across files in an archival store using progressively improved methods. In this chapter we bring together the compression framework with the object storage framework, and an adaptive algorithm to select compression methods to yield high compression of highly redundant data and low overhead for highly unique data.

8.1 Overview

The space efficiency of lossless data compression is highly data dependent. Often the most important compression metric is *space efficiency*: how little memory is required to store encoded data compared to the original unencoded data. Two other metrics are *compression performance* and *decompression performance*, the rates at which data can be converted between uncompressed and compressed forms and vice-versa. Metrics of secondary importance include computer resource requirements and utilization including computation, memory, disk,

and network usage. Resource requirements play an important role in determining the design and implementation of storage systems.

Data compression properties also depend on each compression algorithm's effectiveness in detecting and eliminating redundancy. Unfortunately, the best algorithms today do not work optimally in all respects. For example, the whole-file hashing technique used to detect identical files (with high probability) requires only a small fixed-size block of RAM, modest CPU resources, but reads arbitrarily large amounts of data serially from disk in order to compute an identifying hash value. Digests are computed with high throughput, often higher than disk read/write throughput, but they do nothing to help reduce redundancy seen in similar files. In contrast, binary differencing between all pairs of files could be used to compute the smallest encodings between files, but the computational complexity of order $O(n^2)$ would be too high to make it feasible in a production environment.

A more practical solution exists: progressively apply data compression algorithms which yield high efficiency and high performance first and only when they do not provide a satisfactory result, apply lower efficiency and lower performance algorithms.

PRESIDIO is designed as follows. Each data compression algorithm is modeled to provide an approximate data compression efficiency yield as a function of the resemblance of input data. The properties of the model include expected data compression storage efficiency, the input read rate, and the output write rate. When a candidate file is presented to PRESIDIO, it evaluates the data and compression models to compute scores for each algorithm. The algorithm with the best score is applied. If the result of the data compression achieves the compression threshold for the model for both storage space and performance, the algorithm is used, otherwise the next-highest algorithm is applied. The evaluation process is repeated until the available compression algorithms are exhausted.

The PRESIDIO framework also incorporates a content-addressable store into which objects are stored including whole files, subsections of files, file metadata, and delta files. Every virtual object can be addressed by its *content address* (CA).

Figure 8.1 illustrates the relationships between the different efficient storage methods and the compression methods they use.

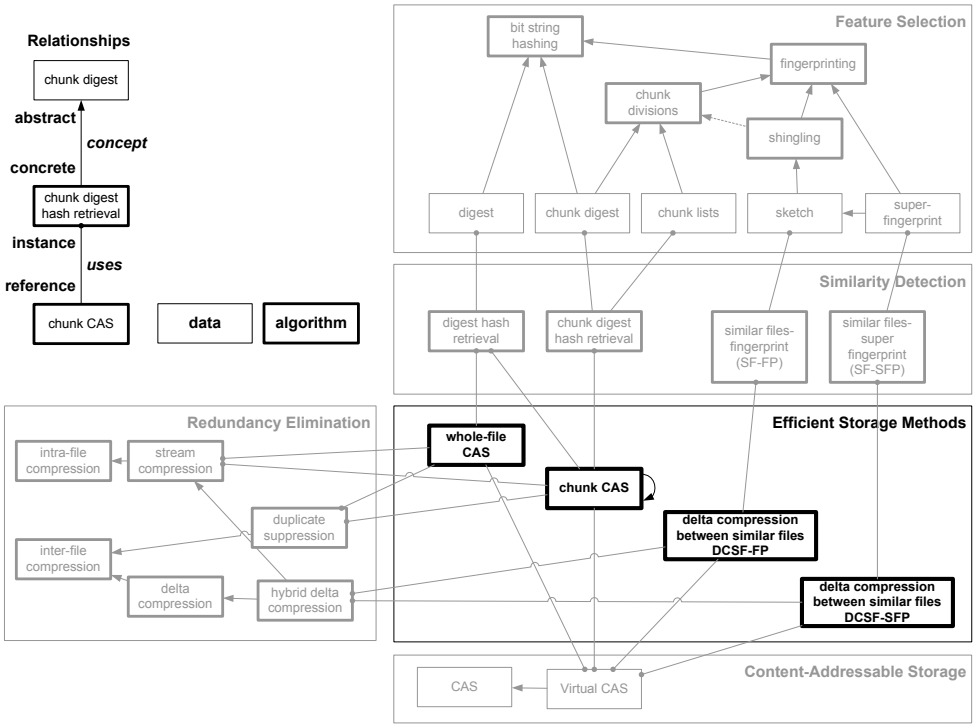


Figure 8.1: Efficient storage methods in the PRESIDIO architecture

This chapter is organized as follows. First, we describe the PRESIDIO storage framework. Second, we detail the PRESIDIO algorithm, the use of efficient storage methods, and the use of the VCAS.

8.2 The PRESIDIO Storage Framework

The PRESIDIO Storage Framework consists of a layered architecture with the following major components:

- the *progressive redundancy eliminator* (PRE) compression algorithm,
- *efficient storage methods* (ESMs),
- *virtual coding methods* (VCMs),
- and a *virtual content-addressable store* (VCAS)

Figure 8.2 illustrates the main storage components, the mid-level PRESIDIO framework, and the low-level VCAS and CAS storage subsystem. Each of these components supports two main storage operations, *store* and *retrieve*.

PRESIDIO presents its storage interface to detect, select, and employ the most space-efficient storage method. Its progressive redundancy eliminator algorithm (PRE) links against a set of known ESM classes; from each class and object instance is created. PRE calls methods in each ESM object in turn, using polymorphic object-oriented to select features and to detect resemblance. Once PRE selects an ESM, it is called to compress a file into the VCAS.

The VCAS storage subsystem is made up of two parts: *virtual coding methods*, which use content addresses to represent virtual but not internal representation, and the CAS, which uses content addresses to represent real content. A low-level VCAS interface is used to write objects into a content-addressable store by encoding them using a small number of *virtual coding methods*. Each method is made up of a *coder* and *decoder*; the virtual coding methods are *codecs*. The CAS implementation we use consists of a simple CAS database and megablock log file store, as described in Section 7.6.

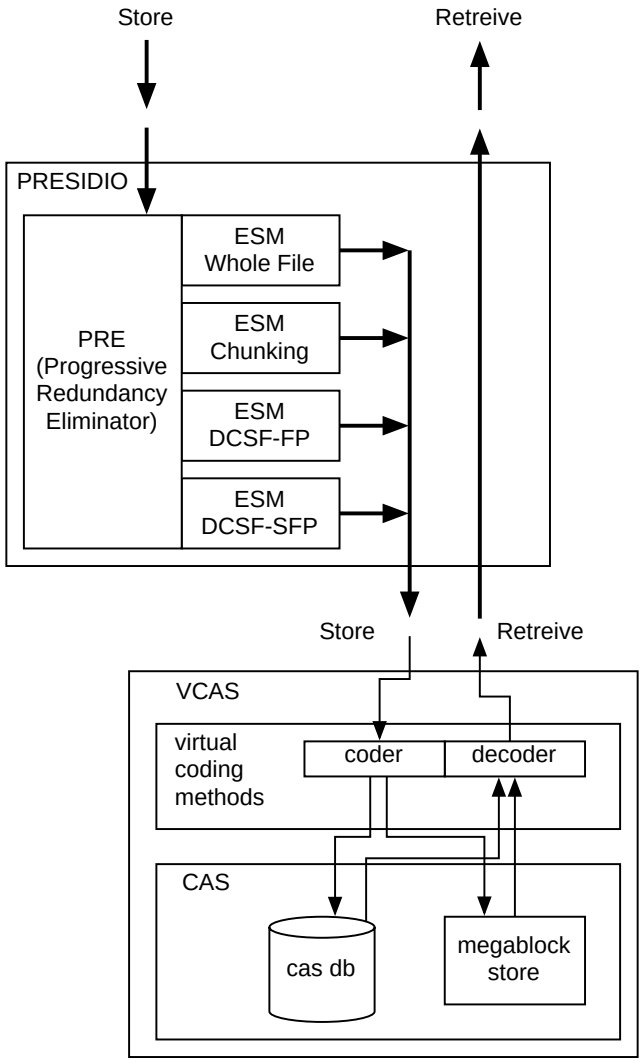
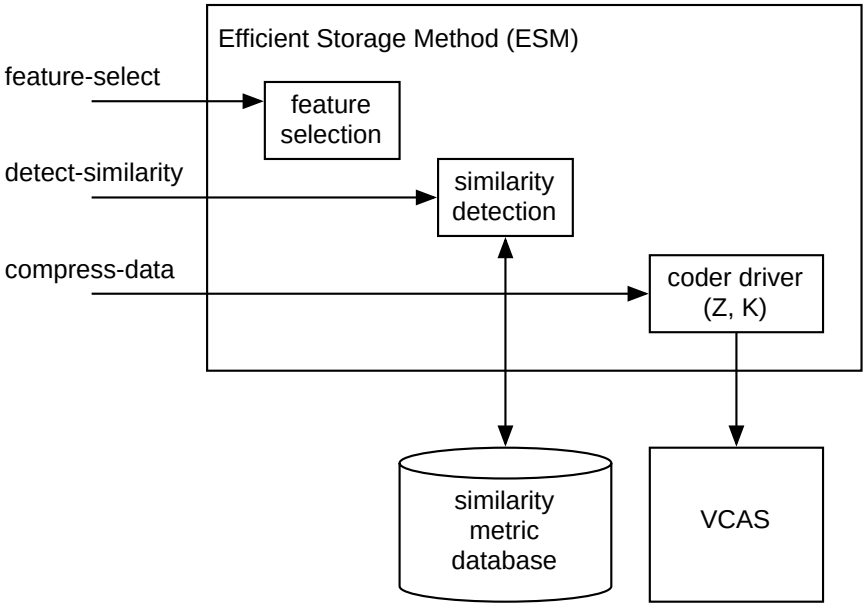


Figure 8.2: PRESIDIO ESM and CAS

There are some notable differences in this design when compared to existing CAS systems. The first is the use of multiple methods for detecting similarity and eliminating redundancy unlike pure CAS systems like Centera that use a single method such as whole-file hashing, or chunk-based hashing. The second is the use of a virtual content-addressable store instead of a traditional CAS. The third is the use of an algorithm to detect and use efficient

storage in a progressive manner. The last is the use of a storage system design that combines multiple compression methods into a hybrid archival storage system.

8.3 Efficient Storage Methods



feature-set = feature-select(file, content-address)

(r, resemblance-state) = detect-similarity(file, content-address, feature-set)

compress-file(file, content-address, feature-set)

Figure 8.3: Efficient storage method operations

The PRESIDIO architecture defines ESMs as a combination of data and algorithms—or simply, a class of objects—whose purpose is to select features from a file, determine the effectiveness of a specific compression method for a given file, and then to marshal the parameters used to drive the input to a VCAS coder. ESMs are expressed as class definitions with the

objective of compressing a single type of CAS compression.

Each ESM implementation uses the same common programming interface made up of three algorithms, implemented as independent or inherited functions. Additionally, some state may be shared between multiple ESMs. The following describes each part:

Feature selection: an algorithm taking a file as input and selecting features from the raw data, returning a feature set as output.

Similarity detection: an algorithm taking a file as input, the file's feature set, and returning *r*, a resemblance metric and resemblance-state which may be used to help encode (compress) the file.

Coder driver: an algorithm to marshal parameters that will be passed to the VCAS to encode a file.

Similarity metric database: an optional persistent state containing information needed to detect similarity of a file against previously stored files.

Figure 8.4 illustrates the different ESMs we have developed. Each ESM box represents a different implementation. Some ESMs may share state, for example in the case of DCSF-FP and DCSF-SFP, the *sketch database* contains fingerprints per file. On the right, the coder driver will call the VCAS with specific parameters. Following we list the C++ declaration for the ESM implementation, **class TVirtualFile** and then describe each of the ESMs in turn.

8.3.1 `class TESM`: the ESM class interface

The ESM interface contains two types of information: a pointer to a polymorphic `TVirtualFile` object and PRE evaluation state. The internal state consists of the following.

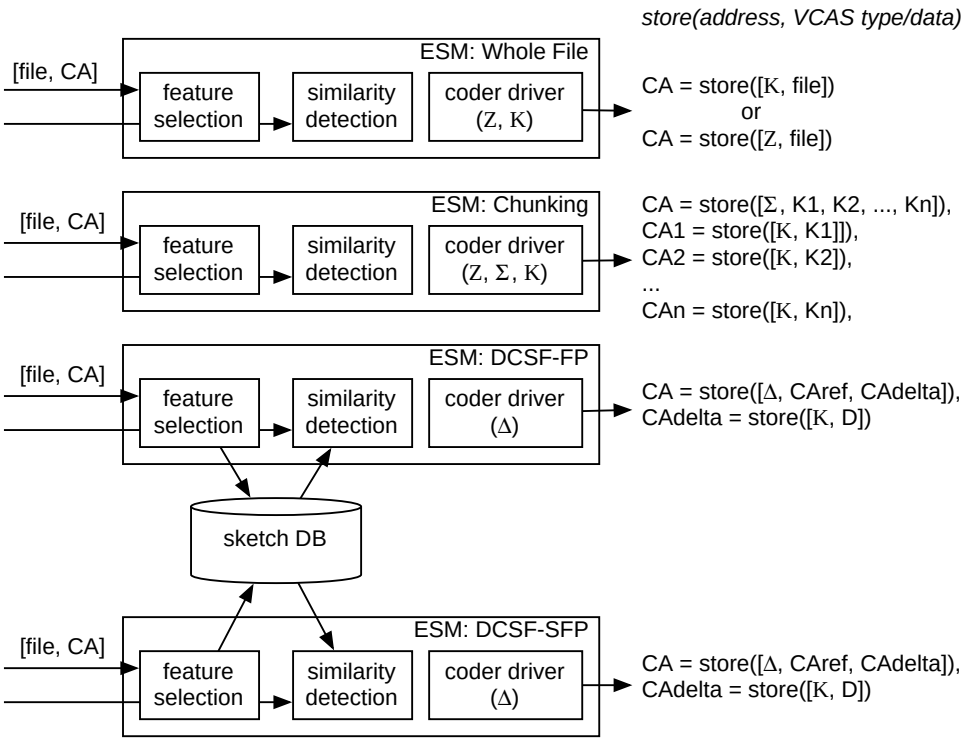


Figure 8.4: PRESIDIO Efficient storage methods

The *virtual length*, l_v, is equal to the length of the file in bytes presented in a *store* request. The *real length*, l_r, is the approximate or exact storage size in bytes of the file, including storage overhead for intermediate VCAS structures.

We define *efficiency* as the incremental storage size of a newly presented storage file as a fraction of the virtual storage size, where 0 is perfect efficiency, no additional bytes stored; and 1 is no efficiency, all input data is stored uncompressed.

PRE scoring is computed from real storage efficiency, u_{real}, and biased storage efficiency, u_{biased}. Bias, b, is a coefficient to modify the efficiency score. Efficiencies are simply:

$$u_{real} = l_r / l_v \tag{8.1}$$

$$u_{biased} = bu_{real} \tag{8.2}$$

Low u reflects data that is *more* compressed.

Bias is a factor to help offset the introduction of storage overhead for ESMs that have the potential benefit but not during initial evaluation. For example, the whole-file hashing ESM will introduce no virtual storage overhead in the VCAS because the entire file can be stored a single CAS object. A chunk-based ESM would introduce a list of chunk CAs, a slight increase in storage. One reason to use bias is if chunks were never stored in the first place, then there would be no opportunity to share chunks later. By introducing bias, slightly inefficient ESMs, *e.g.* chunking, can be selected opportunistically despite that method's higher, non-biased u. Our implementation uses a constant bias, but adjusting bias dynamically is a potential area to further improve storage efficiency.

The interface for `class TESM` is as follows.

```
class TESM
{
public:
                TESM();
                ~TESM();

public:
  TVirtualFile* fVirtualFile;           // alias to file object
  double        fBias;                  // efficiency bias

  size_t        fVirtualLength;         // bytes for source data
  size_t        fRealLength;            // bytes for actual storage
  double        fEfficiency;            // real/virtual length
  double        fBiasedEfficiency;      // bias*efficiency
  double        fDuration[kTotalPhases];
};
```

8.3.2 `class TVirtualFile`: the virtual file interface

The TVirtualFile class embodies the object-oriented implementation for each ESM. We chose to use an object-oriented design (OOD) for several reasons: first, we used *abstraction*

to advance the definition of *protocol* and *data relationships* over implementation details; second, we relied on *class inheritance* to reuse and build on common ESM data and behavior; and third, we used *polymorphism* to allow the PRE algorithm to invoke methods on each ESM object's methods without internal knowledge.

Abstraction allowed us to use classes such as `TContentAddress`, the object representing content addresses, to define its own internal format and size, as well as to allow the object to read and write itself.

Class inheritance afforded method reuse and helped define common object data. For example, raw data and (zlib-) compressed raw data possess the same content address and their whole-file hash features are computed identically, namely a single digest computed over the entire uncompressed file, but their internal real stored object representations are different, namely uncompressed versus zlib-compressed CAS objects.

One shortcoming using C++ as an object-oriented programming (OOP) language is that introspection and enumeration of dynamically defined subclasses is not automatic. So each ESM is listed statically in a small structure over which PRE iterates. Because of the small number of classes and the control a system designer would want to exercise over the specific ESMs used, we do not see this as a practical drawback.

```
class TVirtualFile
{
public:
  TVirtualFile();
  TVirtualFile(const TContentAddress& ca,
               MetaFlags metaFlags = kMetaNone);

  virtual size_t           SelectFeatures(TFile& sourceFile);
  virtual size_t           DetectResemblance(TCAS& cas,
                                             TFile& sourceFile) = 0;

  virtual TContentAddress  Store(TPRE& pre,
                                 TCAS& cas,
                                 TFile& sourceFile) = 0;
```

```
  virtual size_t           Retrieve(const TContentAddress& ca,
                                    TCAS& cas, TFile& destFile) = 0;

protected:
  TContentAddress          fCA;
  MetaFlags                fMetaFlags;
  size_t                   fVirtualLength;
};
```

The main member functions used from this class are:

```
SelectFeatures(TFile& sourceFile)
```

SelectFeatures computes or selects features from an input file and returns the length of the virtual content addressable storage object. The class derived from TVirtualFile instance may retain feature state.

```
DetectResemblance(TCAS& cas, TFile& sourceFile)
```

DetectResemblance computes the best resemblance based on space efficiency and return the exact or approximate real length for the object that would be stored using this method.

```
Store(TPRE& pre, TCAS& cas, TFile& sourceFile)
```

Stores the file object using the ESM method by making calls to the real CAS. The virtual coding method may recursively call the PRE algorithm to further improve storage efficiency.

```
Retrieve(const TContentAddress& ca, TCAS& cas, TFile& destFile)
```

Retrieves an object named by the content address, using the CAS and return its virtual length, writing to the destination file.

8.3.3 ESM: Whole-file

The *Whole-File* ESM implements the standard content-addressable store, corresponding to the schematic in the upper box in Figure 8.4. This ESM has the highest possible compression when content addresses match exactly. When identical files are presented to the ESM, it can easily detect the existence of a file through a hash lookup in the VCAS database quickly. Compression is actually implemented as suppression of any storage operation to the VCAS. The feature selection algorithm returns the hash of the file; our prototype uses the MD5 hashing function.

The similarity detection method is implemented by testing for the file existing in the VCAS using the content address. Efficiency u_{real} is 0 when the file exists or 1 if not. (Resemblance r is binary: $r = 0$ when the file is not found in the VCAS, or $r = 1$, the hash for the file already exists in the VCAS.)

The coder driver function named `TConstantFile::Store` marshals the "constant data" operation, **K**, and the file data **file** using the following pseudocode:

```
esm[i].fVirtualFile = new TConstantFile(ca);
...
// PRE evaluation selects an esm numbered 'i'
...
TContentAddress ca = esm[i].fVirtualFile->Store();
```

In virtually all cases the compressed data size is smaller than or equal to the compressed size, and since the feature selection operation computes into a temporary file, the **Z** VCAS operation (Zlib compression) is used. The specification for the file is identical, except for the following

```
esm[i].fVirtualFile = new TZlibConstantFile(ca);
```

8.3.4 ESM: Chunking CAS

Chunking divides a file into blocks using a deterministic algorithm. The concept allows for a wide variety of file-division algorithms including fixed-sized blocks like Venti [125], or using blocks whose lengths are determined by randomized algorithms like LBFS [110] or DERD [40].

The feature selection is the list of CA for chunks. Our algorithm uses the following parameters, previously described in Table 4.2: $w = 32$, $m = 64$, $M = 4096$, $d = 1024$, $r = 0$, $RF_{size} = 32$, $RF_{poly} = \texttt{0x1A8948691}$, $CID_{size} = CA_{size} = 128$. The feature set that is returned is a sequence of content addresses $CA_1, CA_2, \ldots, CA_n$ corresponding to the chunks $K_1, K_2, \ldots K_n$ such that the concatenation of chunks K_i is the input, **file**.

Similarity detection computes the potential real storage size by iterating over the candidate chunks in two steps. First, for each chunk it determines whether the chunk has already been stored, in which case the incremental real chunk size is zero, otherwise the real chunk size is computed from the similarity detection using PRE. In other words, an entire chunk itself is evaluated recursively for its potential real size. Second, each chunk K_i is added to the chunk lists and then that list is evaluated for its size, using recursive PRE.

The recursive method for similarity detection has one major benefit: if a chunk list has already been stored, or if resemblance can be detected from the chunk list, then it too will be stored efficiently.

Our current recursive implementation statically computes the intermediate results, in effect performing many of the compression steps to compute the actual VCAS real (compressed) object size. Heuristic evaluations of the virtual content may be able to yield qualitatively similar results without incurring the read/write or computational overhead.

The coder driver marshals parameters: a concatenated "chunklist" (Σ) of chunks.

Each chunk is submitted back to PRE so that it can also be stored efficiently, generally either as constant chunks (K) or compressed chunks (Z). In practice, only compressed chunks are written. To record the files, the chunklist is written to the VCAS followed by chunks that have not already been written.

8.3.5 ESM: DCSF-FP

Both the *delta compression of similar files using fingerprints* (DCSF-FP) and *delta compression of similar files using superfingerprints* (DCSF-SFP) start with identical feature sets, but use slightly different similarity detection algorithms. Computing DCSF feature sets is a computationally expensive operation, so it is only computed once. However, once it is determined, it is easy to compute additional features, *e.g.* harmonic superfingerprints, that make similarity detection faster due to a smaller number of feature lookups. Separating the DCSF-FP and DCSF-SFP algorithms into two ESMs makes it possible to share common data and to find highly similar data quickly.

Feature selection computes a sketch of fingerprints. Each sketch is an vector of $k = 32$ features selected from the file using k functions to independently select features from a sliding window of size w in the file using the min-wise independent permutations.

Similarity detection is a comparison between a file with a sketch and all sketches in the database. The comparison function returns resemblance r ($r = max(n/k)$ for all sketches, where n is the number of features that match).

Our implementation does not currently use information retrieval (IR) techniques to retrieve from feature vectors of size k. The implementation can be performed using methods such as inverted word lists, where each word is represented by a fingerprint or by using Bloom filters. In both cases, the in-memory and on-disk usage are high and such overhead may not

offset potential compression gains.

The coder driver writes a delta (Δ) between a reference file (R) and a delta file (Δ); a delta file is also written as constant data (K) and the contents of the delta file itself (D).

8.3.6 ESM: DCSF-SFP

At the time a sketch is computed, the harmonic superfingerprints are also easily computed. Superfingerprint features give high probability of matching sketches with high resemblance using a much smaller set of fingerprints.

DCSF-SFP similarity detection is a comparison between one or more superfingerprints. Instead of comparing entire feature sets (or feature vectors), one or more superfingerprints are compared. When harmonic superfingerprints are used, one superfingerprint covering all features $f_i, 0 \leq i \leq k$ can be indexed directly. Retrieval and comparison of harmonic superfingerprints covering a subset of sketch $S(A)$ require additional indexing and retrieval. The coder driver for the DCSF-SFP is identical to DCFS-FP.

8.4 Content-Addressable Storage (CAS) and Virtual CAS (VCAS)

We use CAS as a common abstraction. We further use VCAS to allow us to further abstract the storage mechanism from the storage interface. VCAS allows us to reconstruct data, it also allows us to rewrite the data; meanwhile CA lets us ensure that the data is unchanged. VCAS also allows us to store more than one copy—useful for caching—and with different coder representations—-useful for keeping alternate versions of cached fully reconstructed files to improve performance.

Content-addressable storage is a basic storage system. Application programming interfaces (APIs) for both CAS and VCAS interfaces use content addresses for identifying fixed

content data. This differs from file system APIs, which create, delete, read, and write files and their content based on a file's location and position within a file. CAS, unlike file systems, use an underlying storage implementation that can include a file system, database, or other storage structures.

Since the goal for our solution is to improve space efficiency, the low-level CAS storage must also be space efficient. We design and measure storage efficiency to account for additional overhead. In particular, per-block overhead and database structures for indexing must be considered. File systems use block-aligned storage to satisfy dynamic allocation and high performance read/write operations, but introduce overhead storage in unused storage at the end of blocks, extents, i-nodes, and volume structures. Cluster file systems must also maintain additional structures for distributed access. Reliable storage also introduces some redundant data. These last two are design considerations for future investigation in the Deep Store project, but the overall design goal to minimize overhead storage still exists.

8.4.1 Virtual Content-Addressable Storage (VCAS)

All file content stored is stored in the virtual content-addressable storage subsystem. CAS objects are stored as either *real data* or *virtual data* and are described in detail in Chapter 7. The architecture of the VCAS incorporates a set of *codecs*, or coder/decoder pairs.

The following describes how the VCAS design solves several problems.

The VCAS is a low-overhead virtual object (VO) storage mechanism. Object overhead is a small number of bytes. The absence of block allocation eliminates internal fragmentation common in many file systems.

VCAS object encodings (VOEs), the metadata that specifies how to reconstruct files, are stored within the VCAS itself. Each VOE is written as a single serialized stream of bytes

that can be stored in a CAS and addressed by its CA. VOEs can also be stored as virtual objects themselves, but in practice they are small (size $\ll$ 1 KB) and stored simple as constant (literal) data.

Polymorphic object storage lets us reconstruct based on methods that are stored with the data. Each VOE is a single method that can be used within a recursive VO definition. VO reconstruction is a recursive reconstruction of the underlying data. The polymorphic behavior exists due to a common definition for a class of objects (VOEs) that defined by concrete implementations that define a specific set of operations. Instances of the objects specify the methods required to reconstruct real objects based on a recursive definition.

VOE coder and decoder methods are separated from the efficient storage methods (ESMs). Once files have been stored, only the decoder implementations are required. To help ensure permanence by lowering software maintenance, the more complex similarity detection, redundancy elimination methods, and efficient storage methods are not needed once the data has been recorded. To further simplify the need for maintenance, only the VOE decoder implementation is needed.

8.4.2 Progressive Redundancy Elimination

PRESIDIO incorporates the *progressive redundancy eliminator* (PRE) algorithm. Its purpose is to use a collection of ESMs to first determine the best method for eliminating redundancy, and then to encode the data into virtual object encodings. Next, the encodings and original file content are passed to the VCAS where the data is written.

In practice, implementation or design improvements such as programming optimizations for the fingerprinting function, clustering or ordering requests for a file, can improve performance The delta encoding method requires more compute resources than chunking and

Algorithm 2: Progressive Redundancy Eliminator (PRE)

initialize *feature selection and resemblance detection*

k ← size of ESM list

for $i = 0$ to $i < k$ **do**

TVirtualFile* vf ← TVirtualFile::CreateVirtualFile(type)

TESM& e = esm[i]

e.fVirtualFile ← vf

e.fVirtualLength ← vf->SelectFeatures(sourceFile)

e.fRealLength ← vf->DetectResemblance(cas, sourceFile)

e.fEfficiency ← e.fRealLength / e.fVirtualLength

e.fBiasedEfficiency ← e.fBias × e.fEfficiency

if e.fRealLength = 0 **then**

break *found maximally efficient file (exact match)*

end if

end for

Start PRE evaluation

bestESM ← 0

bestBiasedEfficiency ← 1.0

for $i = 0$ to $i < k$ **do**

if esm[i].fBiasedEfficiency < bestBiasedEfficiency **then**

bestESM ← i

end if

end for

made up of three main phases: computing a file sketch, determining which file is similar, and computing a delta encoding. Currently, the most costly phase is computing the file sketch due to the large number of fingerprints that are generated. For each byte in a file, one fingerprint is computed for the sliding window and another 20 fingerprints are computed for feature selection. The shingling performance in our initial prototype was approximately 0.5 MB per second (Table 4.4), far less than the approximate 30–50 MB per second disk read performance. Fortunately, through careful programming, we were able to rewrite the fingerprinting and shingling code to improve throughput dramatically, increasing throughput to just under 20 MB per second.

The second operation, locating similar files, is more difficult. Our current implementation is not scalable since it compares a new file against all existing files that have already been stored. Scalability in searching for a similar file is a particularly interesting area of our research. We are also optimistic that large-scale searches are possible given the existence of web-scale search engines that index the web using similar resemblance techniques [15].

Chunk storage and delta compressed storage exhibit different I/O patterns. Chunks can be stored on the basis of their identifiers using a (potentially distributed) hash table. There is no need for maintaining placement metadata and hashing may work well in distributed environments. However, reconstructing files may involve random I/O. In contrast, delta-encoded objects are whole reference files or smaller delta files, which can be stored and accessed efficiently in a sequential manner. But, placement in a distributed infrastructure is more involved.

A couple additional issues exist for delta encoding that are not present with chunking. Because delta encodings imply dependency, a number of dependent files must first be reconstructed before a requested file can be retrieved. Limiting the number of revisions can bound the number of reconstructions at a potential reduction in efficiency. Another concern that might be raised is the intermediate memory requirements; however, in-place reconstruction of delta

files can be performed, minimizing transient resources [23]. At first glance, it would appear that the dependency chain and reconstruction performance of delta files might be lower than reconstruction of chunked files, but since reference and delta files are stored as a single file stream and chunking may require retrieval of scattered data—especially in a populated chunk CAS—it is unclear at this point which method would produce worse throughput.

8.5 Summary

In this chapter we have integrated a unified archival data storage model with algorithms for storing data efficiently in PRESIDIO. The the *progressive redundancy eliminator* (PRE) compression algorithm, applies multiple *efficient storage methods* (ESMs) to find identical or similar data, then employs *virtual coding methods* (VCMs) to record data into a *virtual content-addressable store* (VCAS).

Efficient storage methods are objects which implement three protocols: a method to select features, a method to detect similarity, and encoding/decoding functions to compress data. A virtual C++ file class interface is the mechanism by which these protocols are implemented.

The PRE algorithm computes efficient rank from the likely real storage compression size compared to the (virtual) input file size biased by an adjustment to overcome storage overhead.

Chapter 9

Conclusion

We conclude this dissertation with a summary of our contributions, a short discussion, limitations of our solution, and directions for future work.

9.1 Contributions

To meet the growing need of storing increasing volumes of fixed-content or archival data, the development of archival disk-based systems have improved accessibility and to increased storage efficiency. Immutable read-only archival data offers the opportunity to use content-addressable storage, disk-based storage affords low-latency and random access, and complementary forms of data compression make use of these technologies possible. However, the cost of using magnetic disk is higher per byte than traditional archival media, therefore storage space efficiency in archival disk systems is one of the most important problems to be solved.

Since the introduction of content-addressable storage systems, there have been many developments to continue to improve storage efficiency. Research in bandwidth reduction tech-

niques and algorithms to divide files into smaller pieces allow unique data to be identified and for the storage of duplicate data to be suppressed. These methods are optimal. Subdivided files incur overhead storage cost with potentially no storage reduction, and similar data constitutes potential space efficiency increases. Furthermore, existing data reliability models do not account for increases in failures to large numbers of files that are compounded by data dependency.

Improved storage efficiency is possible by detecting similar data and reducing redundancy by delta encoding versions of files. Similarity detection can require intensive use of resources such as CPU, memory, and disk. These similarity detection algorithms are not necessarily compatible with pure content-addressable storage systems which depend on detection of identical data in order to suppress duplicates.

The difficulty with existing storage systems is that they do not provide optimal space efficiency over all content and do not use a common architecture that allows the efficiency benefits of one compression technique to be used within another efficient storage technique.

The main contribution of this dissertation is to *unify archival storage solutions to maximize space efficiency*. To achieve this goal, we have described the design and implementation of the Progressive Redundancy Elimination of Similar and Identical Data in Objects (PRESIDIO). The premise of the development system is threefold: the storage architecture presents a simple content-addressable storage interface, a unified virtual content-addressable store encodes data using polymorphic methods that use multiple storage methods, and an internal storage framework in which to express and evaluate multiple efficient storage methods.

9.1.1 Problems addressed

With PRESIDIO, we have developed a storage system to address problems that exist across content-addressable and archival storage systems today:

- **Efficient storage architecture.** We described common phases in an archival storage system. In Chapter 4, we listed data identification algorithms and evaluated their use on file content. Chapter 5, describes methods to detect similar and identical data between a new file and a previously stored file. And in Chapter 6, we listed and evaluated two methods for storing data more efficiently by eliminating redundancy across files. Our contribution in these areas have been to identify common behavior in storage systems and divide them into separate phases so they can be used in a common storage and compression framework.

- **Unified CAS.** In Chapter 7, we described a novel technique for storing data using a *virtual content-addressable store* in which virtual data is identified by its content address, and real data is encoded polymorphically.

- **Progressive compression.** Chapter 8 describes the design, implementation, and evaluation of the Progressive Redundancy Eliminator (PRE) algorithm, and its use with VCAS storage to make the PRESIDIO framework.

9.1.2 Technology benefits

Our contributions translate into a number of benefits. Our solution shows that we can improve the storage efficiency of an archival storage system using progressively higher levels of compression when it is available. This makes it possible to consider improved data

compression methods, like delta compression that may show marked efficiency improvements, and still benefit from other cheaper, but less efficient methods like chunking.

The unified content addressable storage system offers unique benefit. The virtual CAS creates a level of abstraction that separates file contents (and file metadata) from internal storage data and storage metadata. Furthermore, the virtual representation presents other opportunities, such as the ability to store more than one representation of a file, *e.g.* both whole file and delta file for reliability, redundancy, and retrieval efficiency. A more significant improvement is that some internal data can be stored in more than one way. For example, chunk lists can be stored as delta-compressed data and vice-versa.

The simple nature of virtual content methods decouple the coder from the decoder. A simple set of virtual encoding primitives mean that long-term data retrieval does not depend on encoding operation. One of harder problems in archival storage is ensuring permanence of logic as much as the permanence of data. By implementing the VCAS simply, we feel it contributes to the long-term viability of the data it stores.

The PRESIDIO framework–and frameworks by their very nature—defines a structure into which additional methods can be introduced. Both VCAS file encoding implementations, as well as PRE analysis methods can be overridden or implemented from scratch. This offers additional opportunity for further research and improvement.

9.1.3 Minor contributions

Some minor contributions include the *chc* chunk compression program, which eliminates chunk-based redundancy within files in a manner that is complementary to stream-based compression. We have evaluated different chunk compression against delta compression to show the relative efficiency of each. In the case of chunk compression, dividing files into

chunks and identifying chunks by hash-based features is a fast operation while delta compression requires similarity detection and delta compression.

We have introduced harmonic superfingerprints, a variation of superfingerprinting that allows progressively higher levels of recall and precision by increasing the coverage of superfingerprints in a feature sketch. Finally, we have implemented an initial version of the PRESIDIO framework that include the VCAS (virtual content addressable store), ESMs (efficient storage methods), and the PRE (progressive redundancy eliminator) algorithm.

9.2 Discussion

PRESIDIO is a departure from the design of traditional file systems. Although similar to some more recent disk-based archival storage systems, the design requirements for Deep Store and the PRESIDIO framework are substantially different from high-performance file systems. The first and most notable difference is the analysis of files. Whereas file systems do not typically analyze files for content, archival storage systems—in particular, efficient archival storage systems—require multiple types of content analysis to detect and eliminate redundancy.

Another significant difference in our work is to abstract a file's contents from the underlying mechanisms from which it is stored. While content may not be the only way to identify data, automatically selecting features and performing similarity detection across many files of arbitrary types creates the opportunity to think about file storage in other ways. Additionally, our design advances the idea storage and data retrieval should be easily separated by creating self-descriptive data structures from which contents can be retrieved.

During our development, we we discovered that there are new ways of visualizing storage behavior, whether it be parametric in nature or measurements in performance. Much work in file systems is focused on increasing throughput or reducing latency. For that, trace

analysis and instrumentation are useful. In our investigations, other tools are needed to help visualize similarity and the relationships between objects. Extensive data dependencies, not common in files in traditional file systems, are prevalent in an environment where inter-file data compression is a means to the goal. While aggregate statistics are useful, many types of investigative questions are not easily answered by viewing numbers or tables; we found two- and three-dimensional representations of dependency graphs and graphs showing similarity or resemblance invaluable. However, with large numbers of data points, many tools did not scale well enough to give both "big picture" and detailed information.

The solution to our problem is different from the design we originally anticipated. One main concern for Deep Store was to store large amounts of data reliably over a large number of storage nodes, but we chose to focus our work on space efficiency over scalable storage. Despite the narrower focus of our problem space, we have kept the design of distributed storage system in mind. In particular, we decided that the flat namespace of a CAS system would permit data distribution more easily by partitioning a hash space, for example by using distributed hashing like LH [87, 89, 88] to partition and distribute the tables.

Another change from our original solution strategy was not to use data clustering of similar data [75, 29, 33, 43, 109, 176]. Although its use might still be feasible, we found that data clustering might not scale well when clusters are grown incrementally by adding new files and their feature sets. In addition, the performance of managing clusters and searching through them would depend greatly on the similarity of the data. For example, highly dissimilar data would produce either many singleton clusters, or clusters with dissimilar data.

Perhaps the most significant realization to us is that our understanding of data storage models has changed greatly throughout the development of our solution. The first discovery is that storing data efficiently is less of a storage or compression problem than an exercise in

system design employing data fingerprinting and information retrieval methods incorporated into an architecture that would meet cost and resource constraints. To this end, we would need to incorporate research and solutions from other disciplines. The second discovery is that data dependency has a significant impact on archival storage whereas it would have little impact in a file system. As is obvious to those making contributions in the data compression field, compression is completely data dependent; this is perhaps more true for large-scale data compression. The third discovery is that disk-based archival data storage is still an open field.

Although we have tried to address one problem, that of space efficiency, many hard problems still exist. Engineering mechanical devices and designing storage reliability for data permanence will continue to be a difficult problem. In addition, satisfying the data permanence problem is increasingly a logic permanence problem. Most data today is encoded in one form or another; our virtual encoding is no exception. Without other mechanisms to ensure correct interpretation of bits in perpetuity, the bits themselves will become meaningless.

The future may even more promise for highly interdependent data storage. Our solution was developed for magnetic disk, but its application may find better use with solid state memories. Disk, even with its lower latency than tape, offered the first possibility for content-addressed storage. However, disk performance is still a limitation when random access is the rule, not the exception. With solid state memories that offer lower latency, interdependent data storage will be more attractive due to the lower random access cost. With this, PRESIDIO would offer higher reconstruction throughput and potentially higher performance for retrieval of feature data that are used for similarity detection. In this case, low latency could open further opportunity for additional gains in storage efficiency.

9.3 Limitations

Our work is limited in a number of ways. Data compression is an information theory problem in which an optimal solution may be intractable. Our solution uses many heuristic techniques and engineering compromises to detect data similarity. Feature selection parameters that improve resemblance detection through smaller window sizes or larger feature sets increase the probability of detecting similarity with better accuracy, but at an increased storage overhead cost that conflicts with the primary goal of space efficiency. Parameters for chunk size are a compromise—some parameters for one data set may not be as efficient for another. In order for a single storage system to be used consistently for all input, we select one set of parameters, such as fingerprint size, window size, or expected chunk size parameters.

We use metrics that provide approximate storage improvements. Because no *a priori* tests can be applied to data compression techniques to determine its efficiency, an optimal solution demands comparing multiple possible storage encodings; in the case of inter-file compression, resources are not sufficient to afford these comparisons. Because evaluation may require reads of full files, operations like delta compression may be expensive. Our solution favors the use of redundancy elimination with lower resource cost and higher efficiency over higher cost methods that provide lower efficiency gains.

Like other forms of data compression, our engineering solution is restricted to storage, computing, and bandwidth resources that are not infinite. We used data fingerprinting algorithms that are efficient for our purpose, within a single-machine storage unit. We constrain our solution to use minimal primary and secondary storage. We have used data structures that are linear in the size of the input, and a simple database structure that provides low operational complexity for retrieving feature and content data quickly. Computing and selecting features that provide a high quality representation of internal data within a file can be computationally

expensive, but in order to provide sufficient throughput, we needed to limit feature sizes and also improve the implementation. The PRE algorithm uses additional storage read bandwidth because the multiple phases and multiple efficient storage methods (ESMs) cause some amount of re-processing, although much less than if separate storage systems evaluated and stored data independently.

Our solution is not an entire large-scale distributed storage system, but a foundation subsystem designed for a single machine. The simple CAS interface abstracts the requirements for input but may also reduce the ability to redistribute internal low-level data across storage service on other services. In addition, we do not provide distributed hashing or data placement algorithms. However, we developed the architecture with distributed storage as one of the primary goals, and have attempted to not preclude the design for a larger system. In particular, unlike many storage systems which impose structure with the storage of content, ours does not make such association. The virtual content-addressable storage system offers further opportunity to reconstruct data by separating the virtual representation of a file from its real representation.

9.4 Future Work

The solution we have put forth is a starting point for further research and exploration in archival data storage. The thesis of storage large volumes of data efficiently within a single content-addressable storage node is a critical part of the larger problem of storing data within a distributed storage system. Our content-addressable storage architecture does not rely on the metadata used to organize storage systems. It was the goal of this work to fit within the larger goals of the Deep Store project, a large-scale archival storage system.

9.4.1 Distributed Storage, Replication, and Reliability

Distribution and replication are two major areas of research that would extend the existing storage system. Unlike hierarchical and directed acyclical graph directory organizations, our architecture decouples naming and addresses from content, making it possible for data to be distributed widely. In addition, the architecture is agnostic to file content and metadata format, allowing different system and programming interfaces to be layered on top of the system. Distributed file storage, using distributed hashing or peer-to-peer architectures, can be built on top of PRESIDIO.

Distributed file storage introduces additional problems. In order to scale, a system would grow ideally with low overhead as more storage nodes are added to increase storage capacity and workload. Locality of reference—which is not inherent in data organized by content address—would need to improve in order to reduce data dependencies across many storage nodes. In addition, individual storage nodes incorporating PRESIDIO would then be able to refer to data on other storage nodes, so reducing inter-node references and minimizing reconstruction costs would need to be considered.

Data reliability in an efficient storage system will require new ways of thinking about reliability models. Reliability models which assume probabilities of failure per byte or per block do not address changes when data dependency in shared chunks or delta compressed data. When compounded with distributed data storage, the reliability models may be difficult to describe. Using cluster storage like Ursa Minor [2] that provide per-data object reliability settings address this somewhat, but do not currently express data dependencies as a requirement. Replication is not the only possibility; *erasure coding* offers additional benefits such as availability and data integrity over a distributed system [167]. Additionally, distributed archival systems may need to ensure confidentiality and verifiability over long periods of time [171].

9.4.2 Improving Compression

As has been the case with other areas of data compression research, more efficiency can be gained if properties of content type that is being compressed is known. Our system aims treats all binary data without discrimination but doing so may yield further storage benefits. Feature selection of well-known features such as file name, file type, file extension, or *n*-grams can further aid similarity detection.

For example, *BSDiff* [121], a platform-independent binary differencing program for executable programs produces patch files less than half the size of *xdelta* and comparable to platform-specific programs. One shortcoming of introducing arbitrary or extensible data formats to an archival system is the increased reliance on programs whose existence must be ensured for future retrieval.

Recent work to detect similarity of non-textual data [93] transform images into compact data structures to reduce the search space; although its strategy is intended for a particular domain of problems, the methods used to improve the performance of searching may be applicable to a general-purpose similarity search mechanism that can be used within PRESIDIO.

A shortcoming of an extensible system is that the ability to decompress programs in the future also depends on the existence of the software, operating environment, and hardware required to execute the decompressors. This argument has motivated the development of the Adobe Acrobat PDF/A archival standard away from extensibility and towards a common set of standards [85, 67, 119].

File formats for text can be encoded in one format and transcoded into another easily with little or no loss of fidelity. Large collections of textual data created today in native language encodings such as ISO 8859-1 (Latin-1 for Western European languages) or JIS X 0208:1997 (Shift-JIS for Japanese) might eventually be stored in the ISO 10646 (Unicode) standard. If file

differences are stored as transcoding operations, then alternative version of a file can be stored and retrieved with very little space overhead.

9.4.3 Performance

Store and retrieve are the two primary operations of the PRESIDIO storage framework. Storage performance is measured by the rate of ingest, and retrieval performance is measured by both latency and read bandwidth. Our solution is not required to provide append or rewrite semantics; all files are written in their entirety and immutably. Consequently, file store requests can be made asynchronously of retrieval requests. The rate of ingest is still dependent on throughput when the request rate for data is higher than the internal rate of ingest. Thus, the PRESIDIO steps to identify and detect similar data must also satsify the high throughput requirement.

Searching for data may be resource intensive, either computationally or in memory usage, when high dimensionality searches are used. Indexes and inverted indexes are memory inefficient. Our initial solution was to use low-dimensional spaces—for example, through harmonic superfingerprinting—to keep search complexity low. Other methods may help reduce search, for example with multidimensional extendible hashing [117, 116], or with approximate nearest neighbors [72].

File reconstruction requires that dependent data is first reconstructed. As we have shown, efficiently stored delta-compressed data form dependency chains can be longer than one. Reconstruction of any delta would require that dependent data would require data reads and writes of order $O(ln)$ where l is the average chain length and n is the size of each file, as opposed to reading or retrieving a file of order $O(n)$. However, some reference files may also exhibit high degrees of dependence and they may be cached. Due to the sizes of delta

files being small in practice, in-place reconstruction of data may require much less than $O(ln)$ intermediate storage and computation, as shown in earlier work [22, 23]. File dependencies imply direct dependencies and file prefetching would also aid performance [81].

9.4.3.1 File Aging and Access Patterns

File access patterns for file systems have been measured [52] and repeating the experiments for archival storage would help future designs. Gibson *et al.* profiled on-line disk systems, which are likely to have different characteristics from long-term archival storage. An attempt to reproduce the experiments faces a number of challenges, most notably that measuring accesses with accuracy would span years, generations of systems, and the growth and migration of data. Archiving serves a wide range of purpose, as is evident from government records management guidelines [143] as well as the variations in regulatory compliance which have been driving many of the archival storage industry.

Appendix A

Rabin Fingerprinting by Random Polynomials

Rabin fingerprinting by random polynomials [127] (or simply *Rabin fingerprinting*) is a useful method for computing collision-resistant hashes over binary strings. It is used widely in the research literature of content-addressed and archival storage but technical details of its use are often omitted. Rabin's original technical report and Broder's note on applications of the method [11] are the primary works from which an implementation may be developed. During development of our fingerprinting library, we discovered technicalities that were important to understand before we could fully realize the benefit of the method and deploy it in our system.

The two main shortcomings with Rabin fingerprinting are that there is no common specification from which to implement different libraries that produce identical results, and that important implementation details have been omitted in the literature. Our goal is not to develop a specification here, but to provide key design decisions from which to create one.

Unlike functions that hash strings of arbitrary length, such as MD5 [133], SHA-1 [113] and derivatives SHA-256, SHA-384, and SHA-512 [114], the parameters and assumptions used to implement Rabin fingerprinting functions are underspecified in the literature, often only mentioning the size of data type used to store the fingerprint. We describe the Rabin fin-

gerprinting assumptions and parameters more explicitly so that a practitioner could establish a specification with little or no ambiguity.

Another shortcoming of Rabin fingerprinting is the lack of implementations and implementation advice that can improve execution performance dramatically. Differences in performance can make are significant enough to make a program correct but unusable, due to the large number of fingerprint computations necessary for analyzing content. Also, the *Rabin fingerprint* function *per se* is simply a single function; in practice, we have found that a library of related functions is more useful.

This appendix is organized into the following sections: first, a summary of properties of the Rabin fingerprinting method; second, a description of the representations of string (data) and fingerprints; third, implementation details; and fourth, related work.

A.1 Properties of the Rabin fingerprinting method

The basic Rabin fingerprinting by random polynomial function computes a hash of a fixed size from a binary string of arbitrary length [127]. The functions are of a class of *randomized functions* that exhibit uniform distribution of results when selecting a function at random from the set [31].

Broder identified many useful properties of Rabin fingerprints, which offer significant benefit in practice [11]. Many instances of hash functions for a selected fingerprint size can be selected at random to yield a set of of randomized hash functions, a useful mechanism for selecting features from a large file. Also, operations for fingerprinting a binary string can be composed of functions that fingerprint a byte or a word at a time by precomputing tables for an instance of a hash function. And another useful property is the low computational complexity and resulting efficiency computing fingerprints over sliding windows into strings and

fingerprints of concatenated strings.

A.2 Modulo Polynomial Fingerprinting

The Rabin fingerprint function is simply the following expression:

$$f(A) = A(x) \bmod p(x)$$

where $f(A)$ is the fingerprinting function, an irreducible polynomial of degree k over the finite field $GF(2)$.

A.2.1 Binary Polynomials

We use the term *binary polynomial* to describe the specific case of using the finite field $GF(2^n)$ where $n = 1$. $A(x)$ is a polynomial of arbitrary degree whose coefficients are in $\mathbb{Z}_2[x]$ and $p(x)$ is an irreducible polynomial of degree k, $p(x) \in \mathbb{Z}_2[x]$. Modulo polynomial arithmetic operations, including operations modulo an irreducible polynomial, are simple enough to be easily implemented in hardware and have been used in algebraic coding for many years [7]. We describe how this affects the implementor of a fingerprinting library.

1. A fingerprint is computed by taking a bit string $A(x)$ and computing the *residual* by taking A modulo an irreducible polynomial $p(x)$.

2. The fingerprint is represented by a polynomial $f(A)$ of degree k and whose coefficients are in $\mathbb{Z}_2[x]$, *i.e.* the coefficients are from the set of (0, 1), and each one can be stored in a bit. So the fingerprint representation is simply a bit string of $k+1$ bits representing the $k+1$ coefficients of $f(A)$.

3. Like $f(A)$, polynomials $A(x)$ and $p(x) \in \mathbb{Z}_2[x]$ and therefore directly representable as binary strings.

4. A polynomial $p(x)$ of degree k can be represented by $k+1$ bits. Note that the argument x is not evaluated by substituting its value into the polynomial $p(x)$. Instead, the fingerprinting operation is evaluating one polynomial $A(x)$ modulo another polynomial $p(x)$.

5. Similarly, $A(x)$ is a polynomial of degree l and represented as $l+1$ bits. Or in other words, the coefficients to polynomial $A(x)$ is simply the bit string to be fingerprinted. In a computer that stores data in octets (bytes), l is an integer multiple of eight.

6. The modulo polynomial operation is the *residual* from dividing polynomial $A(x)$ by $p(x)$. It is not the same as the arithmetic modulo operation in which the residual (or *remainder*) is the remainder from integer division. However, addition and subtraction are modulo arithmetic, effectively using the exclusive-or bitwise operation for both addition and subtraction instead of integer arithmetic and subtraction. Good descriptions of the operations can be found in elsewhere [7, 102].

7. The polynomial $p(x)$ is *irreducible*, which is a property described below. It may be selected at random from a large, but finite set of irreducible polynomials of degree k. When non-irreducible polynomials are used, hash results have non-uniform distribution and are therefore not collision-resistant.

A.3 Properties of Rabin fingerprinting functions

Rabin functions are a type of randomized hash function that can be used to digest strings. They possess collision resistance, but not cryptographic invertibility. They are easily

represented in binary form.

A.3.1 Randomized Functions

Rabin fingerprinting functions are known as *randomized functions*. Randomized functions are useful for application over arbitrary data to produce results with uniform distribution. A random function can be selected from a set of functions, such as the set of all irreducible polynomials of a fixed degree. The selection process ensures low probability any adversary can provide input data to produce non-uniform distribution of output results. In the scope of our work, we randomly select an irreducible polynomial to create a new fingerprinting function. Once a fingerprinting function is selected, it is retained to produce deterministic results.

A.3.2 Collision Properties

The usual collision properties for hashing functions apply to Rabin fingerprinting functions. Let $p(x)$ be an irreducible polynomial, and $A(x)$ and $B(x)$ represent polynomials whose coefficients can be represented by binary strings. Let $f(A) \bmod p(x)$ be the fingerprint over string A. Then

$$f(A) \neq f(B) \rightarrow A \neq B$$

$$Pr(f(A) = f(B) \,|\, A \neq B) \quad \text{is very small}$$

Broder distinguishes between fingerprinting and hashing. A more detailed description can be found in Section 4.3.

A.3.3 Binary Representation

An instance of a Rabin fingerprint function $f(A)$ is determined by its specific irreducible polynomial $p_i(x)$. Each polynomial has a constant binary representation that can be stored in a file, declared as a constant in a source code file or generated dynamically. A general $f(A) = A(x) \bmod p_i(x)$ can be implemented for arbitrary i.

A.3.4 Concatenation

Rabin fingerprints exhibit the nice property that fingerprints of concatenated strings can be easily computed from the fingerprints of substrings, specifically:

$$f(concat(A,B)) = f(concat(f(A),B))$$

A.3.5 Number of Hash Functions

For any given degree, there is a large number of irreducible polynomials. Rabin [126] states that the density of monic polynomials in $Z_p[x]$ (where $p = 2$ for binary irreducible polynomials) is approximately $1/k$ where k is the degree of the polynomials. This means that with k bits of representation, there are $2^k/k$ hash functions.

Practically speaking, $k = 2^w$, a power of two, to easily fit into a computer memory word. We used $k = 32$ ($w = 5$) and $k = 64$ ($w = 6$). The number of hash functions are $2^k/2^w = 2^{k-w}$; for $k = 32$, this is 2^{27} and for $k = 64$, 2^{58}.

This compares favorably to the Karp-Rabin string search, which matches substrings by hashing using modulo *integer* arithmetic. With integer arithmetic, prime number density is approximately $1/log(k)$ for words of size k, but there are $1/k$ irreducible polynomials. An additional disadvantage to integer modulo arithmetic is that it is not easy to compute the integer

modulo result without extending to words of $2k$ in size. This makes Rabin fingerprinting by random polynomials more efficient because its implementation requires registers of size k and subsequently only requires register-memory datapaths of size k.

A.3.6 Invertibility

Unlike cryptographic, or "one-way," hashes, the Rabin fingerprint is easily inverted due to polynomial arithmetic. For detecting similar data with fingerprints, we rely on collision-resistance; for addressing by content (content-addressable storage) we rely on cryptographic properties to make forgery difficult.

A.4 Data representation

A.4.1 Representing Polynomials

Polynomial representation may be chosen arbitrarily, however since the "Rabin fingerprint" is used widely, it is useful to define conventions that can be applied to its use consistently. The endianness, bit ordering, use of the high-order degree, and prefixed bitstrings will all have an effect on the real representation of fingerprints.

A.4.1.1 Endianness

Memory, such as RAM, is addressable as binary terms as bytes or words. Polynomials whose coefficients are binary terms, can be represented in arbitrary ways, but typically either a *big-endian* or *little-endian* convention is used. Our implementation uses *big endian* polynomials *i.e.* `0x8000000000000007` (64 bits of coefficients) represents

$$x^{64}(implied) + x^{63} + x^2 + x + 1$$

A.4.2 Fingerprint of the "null string" or "empty string"

The fingerprint of the "empty string" depends on the convention used to store bits. We define a convention for interpreting strings that are all zero. The string to be fingerprinted, $A(x)$, and the fingerprint $f(A)$ can be treated in a similar manner in which their high-order coefficients are implied.

An in-memory representation of all zeros can be represented as a polynomial. Two common interpretations are possible:

1. all coefficients are represented by bits in memory, and

2. only low-order coefficients are represented by bits in memory

In the first case, each bit memory represents one coefficient, a_i. This convention has two shortcomings: strings of all zeros make it impossible to distinguish bit strings by their polynomials, and an extra bit is used to encode the high-order coefficient. The first problem is that a bit string "0" would hash to the same hash value as "00000000" and every other bit string with many zeros. Bit strings with many zeros are common, so hash collisions would occur often. The second problem is that computing a Rabin fingerprint with a polynomial of degree $k = 31$ (which fits into a memory word of $k+1$ bits, *e.g.* a 32-bit value) results in a fingerprint of degree $k-1 = 30$, or 31 bits of storage (and the high-order bit is zero). So in effect, the output of the hash function is reduce by one bit unnecessarily.

In the second case, each bit in memory represents a low-order coefficient and the high-order coefficients are implied. Broder's definition is the simplest case, the high-order coefficient for a polynomial $A(x)$ of degree is simply $a_k = 1$, where k is the degree of A. There is no in-memory representation for $a_k = 1$; it is simply a one-bit ghost prefix implied for all computation. In the more general case, it is possible to use an arbitrary polynomial, or n-bit

ghost prefix.

There are three easily defined memory representations for all strings:

high-order bit = 0	$[a_{32}=1][a_{31}][a_{30}]\ldots[a_0]$
high-order bit = 1	$[a_{32}=0][a_{31}][a_{30}]\ldots[a_0]$
arbitrary prefix string	$[PREFIX][a_{31}][a_{30}]\ldots[a_0]$

where $[a_i]$ is a bit, and $PREFIX$ is a bitstring of arbitrary length. In the first case, strings with an arbitrary number of zero bits are indistinguishable from each other. Clearly this is inadequate as zeros are commonly found in binary data stored on disk.

A better interpretation of the binary string is to use that given by Broder [11] which is that all strings are prepended with a "1". What this means is that the "empty string" does not evaluate to have a fingerprint of zero. The Broder definition of an empty string should always have a high-order bit of 1. Then the fingerprint of the empty string would be

$$f(emptystring(x)) \equiv 1$$

Note that the fingerprint for the empty string can be anything, using all 64 bits of coefficient. Alternatively, an arbitrary string can be used to be the "header" or "prefix" that is prepended to each binary string. This only needs to be computed once using the "extend" function.

In this last case, we use a convention

$$A(x) = concat(PREFIX(x), M(x))$$

which allows us to prepend an arbitrary string so as to seed every string with a fingerprint. In implementation, this is identical to initializing a fingerprint so that the "empty string" has fingerprint

$$f(0) = f(PREFIX(x))$$

and not zero. This eliminates the equally degenerate case when a superfingerprint is computed:

$$g(F) = F(x) \bmod q(x)$$

$$q(x) \text{is an irreducible polynomial}$$

$$F(x) = f(x) = f(A) \bmod q(x)$$

where the fingerprint $f(x)$ is then reinterpreted as a binary string $F(x)$.

We return to the original question of determining the fingerprint of an empty string. It is simply the fingerprint of a bit string of the prefix. If the prefix is zero, then the fingerprint is zero. If the prefix is 1, then the fingerprint of $A = PREFIX$ is

$$f(A) = A(x) \bmod p(x) = 1 \quad \text{when} \quad A(x) = 1$$

Other interesting properties fall out from this. If a word is zero, and $f(PREFIX) = 1$, then the fingerprint will be equal to the irreducible polynomial itself.

A.4.3 Implications of the Prefixed String

If strings are not prefixed implicitly before the fingerprint is computed, the fingerprint of any polynomial $f(x) = 0$ would always be 0. So a single byte of zero or a one-megabyte file with all zeros would have the same fingerprint!

The empty string is actually just a specific instance of a more general case of fingerprinting a string that contains a prefix string within it. When fingerprinting a string that just contains the prefix itself, the fingerprint will be 1. This is somewhat degenerate in that repeated sequences of the prefixed string can cause a fingerprint to be produced for the following case:

$$f(concat(PREFIX, PREFIX, PREFIX)) = f(PREFIX)$$

A.4.4 String Concatenation with Prefixed Strings

Suppose we want to concatenate the strings "A" and "B" that contain hidden prefixes. To do this, one must first compute the fingerprint for A, the fingerprint for B, and then as a shortcut, we can use the expression $f(A)x^t + f(B), t = bitlength(B)$. However, the fingerprinting functions are really computing $f'(concat(A,B))) = f(concat(PREFIX,A))x^t + f(B)$. Note that the fingerprint that is computed for B (using extend, for example) will really be $f(concat(PREFIX,B))$. To correctly compute the concatenated string, we first compute each part separately:

$$f'(A) = f(concat(PREFIX,A))$$

$$f'(B) = f(concat(PREFIX,B))$$

Then we need to combine the strings to effectively compute

$$f'(concat(A,B)) = f(concat(PREFIX,A,B))$$

and not

$$f'(concat(A,B)) = f(concat(PREFIX,A,PREFIXB)).$$

So to do that the concatenation function must remove the prefix from B first. In order to do that, the fingerprint for the prefix is simply subtracted from the fingerprint. (Note that we are using modulo polynomial arithmetic in $\mathbb{Z}_2$, so $f(x) = -f(x)$ and $f(x) + g(x) = f(x) \oplus g(x)$) where $\oplus$ is the *exclusive-or* binary operation.

A.4.5 Sliding Window with Prefixed Strings

When computing fingerprints in a sliding window, two steps must occur: first the old word must be removed from the string (and respectively updating the fingerprint), leaving

a fingerprint that appears to be the string with the oldest word removed from the string, *i.e.* $windowsize = 3$, $A0$ is a word (conveniently the size of the fingerprint)

$$A = concat(PREFIX, A0, A1, A2)$$

then to remove $A0$ from the front of the string:

$$A' = concat(PREFIX, A1, A2)$$

and append $A3$ (note: this is exactly the same operation as extending by one word)

$$AA = concat(PREFIX, A1, A2, A3)$$

To remove the presence of substring $concat(PREFIX, A0)$ from the fingerprint,

$$f(A) = f(concat(PREFIX, A0, A1, A2)$$

$$t = x^{windowsize-1} \text{word size, in bits}$$

$$f(A') = f(A) - f(concat(PREFIX, A1))x^t + f(PREFIX)x^t$$

A.5 Constructing a Rabin fingerprinting library

The goal of a Rabin fingerprinting library is to compute fingerprints (polynomials) over arbitrarily sized polynomials. To a programmer, this is simply to compute fixed-sized binary fingerprints over fixed- or variable-length binary data. The representation of polynomials map directly to binary representations and vice-versa. Other parameters, such as the degree of polynomials, must be specified in order to retain the determinism that is necessary of fingerprints.

Computing fingerprints efficiently is also important. Although it is straightforward to compute fingerprints using polynomial operations, with today's microprocessors, it is still

essential to improve performance by using precomputed tables such that fingerprints can be computed by bytes or words at a time.

Specialized fingerprinting operations, such as computing a fingerprint over a single word, might be used many times, as in the case of certain feature selection operations. Writing operations to precompute can dramatically improve the analysis of data.

A.5.1 Computing Irreducible Polynomials

We list some of the steps necessary to implement the function to compute random irreducible polynomials.

Random Numbers The pseudo-random number generators vary greatly in quality. For example, the FreeBSD (4.7) of rand() does not generate very random numbers in the low order bits so using higher-order bits and shifting them seems to make a big difference.

```
assert(RAND_MAX >= 0x00ff0000);
// use higher-order bits for more randomness, e.g. in FreeBSD
return ((rand() & 0x00ff0000) >> 16);
```

Testing for Irreducibility Rabin described a test that can be applied to polynomials to determine their irreducibility [126]:

> LEMMA 1. Let $l_1, \cdots, l_k$ be all the prime divisors of n and denote $n/l_i = m_i$. A polynomial $g(x) \in Z_p[x]$ of degree n is irreducible in $Z_p[x]$ if and only if
>
> $$g(x) | (x^{p^n} - x),$$
> $$(g(x), x^{p^{m_i}} - x) = 1, \quad 1 \leq i \leq k$$
>
> where (a, b) denotes the greatest common divisor of a and b.

Note that for Rabin fingerprinting, $p = 2$. A similar algorithm, which is fairly easy to implement, is described in the Handbook for Applied Cryptography [103], Algorithm 4.69, p. 155. "Testing a polynomial for irreducibility.":

Algorithm 4.69

$p = 2$ [p is a prime, we use $p = 2^1$]

polynomial: $x^m + a_{m-1}x^{m-1} + \cdots + a_0$

this is a monic polynomial (leading coefficient for x^m is 1)

m = degree of the polynomial in $\mathbb{Z}_2[x]$

1. set $u(x) \leftarrow x$
2. for i from 1 to $floor(m/2)$
   2.1 compute $u(x) \leftarrow u(x)^p \bmod f(x)$
   2.2 compute $d(x) = gcd(f(x), u(x) - x)$
   2.3 if $d(x) \neq 1$ then return "reducible"
3. return "irreducible"

Invariants of Irreducible Polynomials The lowest coefficient of (binary) irreducible polynomials is always 1. The number of non-zero coefficients for (binary) irreducible polynomials is always odd. It is easy to count the number of bits in the binary string representing an irreducible polynomial to determine if the high-order coefficient is implied. For example, an irreducible polynomial of degree $k = 32$ would be represented by 33 bits, but because the polynomial is always monic. The high-order coefficient a_0 (in the term a_0x^{32}) is always 1 and therefore may be omitted. However, the number of expressed coefficients is then always even. In the implementations of Rabin fingerprinting functions that use polynomials of degree $k-1$, one less bit is used, which diminishes the range of the hash function, *i.e.* instead of 2^k possible hash values, only 2^{k-1} are returned. In practice, testing for these two invariants are helpful for determining the correctness of the function for testing irreducibility.

Density of Irreducible Polynomials Rabin proves the density of irreducible polynomials of degree k is approximately $1/k$ [126]. Applying a simple statistical test for sufficiently large sample size helps validate whether the test for irreducibility generates an approximate density.

Primitive Polynomials Primitive polynomials are a subset of irreducible polynomials.

Endianness When determining a convention for computing polynomials, choosing an endianness for the representation of the coefficients has a small impact on the computation performance. Choosing a convention is necessary since a requirement of the archival storage system is to be able to reproduce the computation of the fingerprints regardless of the endianness. Handling cross-platform issues with respect to endianness is not a new problem, but in particular this is not just a matter of "byte-swapping" (which is usually how endian-neutral implementations interpret data) since depending on how the coefficients for a polynomial are written, bit strings may be written in either direction.

The byte ordering of a processor architecture will have a small impact on the speed of computing fingerprints. Once a byte orientation is chosen, *i.e.* big-endian or little endian, and the algorithm for computing fingerprints is selected for optimal computation when sliding over words using that byte orientation, then computing fingerprints on processors using the "other" endianness will be more expensive.

Other implementations code chooses to use a little-endian orientation, which is most useful on Intel processor architectures, which are also little-endian machines. Using this convention, in any binary word, the highest order coefficients for a polynomial (coefficients in $\mathbb{Z}_2$) are stored in the lowest order bits. One can picture the binary coefficients in memory or in a register in descending order of magnitude by writing the memory or register backwards from the normal convention (in the sense that computer memory is normally written most-significant bit to least-significant bit from left to right):

Register	b0 b1 . . . b63
Memory, as bytes	b0 b1 . . . b7, b0 b1 . . . b7
Memory, as 64-bit words	B0 B1 . . . B7, B0 B1 . . . B7

Where "b*n*" is bit *n* and "B*m*" is byte *m*.

Bit operations to test individual bits and to shift bytes or words are done opposite of the conventional operations. For instance, in the C language to test whether the "1" (lowest order coefficient) is set, the word size must be known and then the last position in the word must be tested, *e.g.*

```
#define WORDSIZE 64
if (f & 1 << (WORDSIZE-1))
        ...
```

Or to multiply a polynomial f(x) by x to raise the terms by one degree, the coefficients would normally be "shifted left." However since the binary terms are all written in reversed order, they must be shifted right instead, *e.g.*

```
/* multiply f(x) by x */
f >>= 1;
```

One advantage to big-endian is that the binary operations on bytes and words are identical to language operations. We choose to use big-endian because the bit representation for a polynomial is easier to understand.

A.6 Related work

Rabin fingerprinting by random polynomials uses a specific instance of the Galois field $GF(2^n)$ where n is 1. When fingerprints of fixed size that are also a power of two are used, it is possible to use the more general field $GF(2^n)$ [90].

Bibliography

[1] Google Desktop. http://desktop.google.com/.

[2] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: Versatile cluster-based storage. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*, pages 59–72, San Francisco, California, Dec. 2005. USENIX Association.

[3] A. Adya, W. J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, Massachusetts, Dec. 2002. USENIX Association.

[4] M. Ajtai, R. Burns, R. Fagin, D. D. E. Long, and L. Stockmeyer. Compactly encoding unstructured inputs with differential compression. *Journal of the ACM*, 49(3):318–367, May 2002.

[5] B. S. Baker, U. Manber, and R. Muth. Compressing differences of executable code. In *ACM SIGPLAN Workshop on Compiler Support for System Software (WCSS)*, pages 1–10, Apr. 1999.

[6] T. Batu, F. Ergün, J. Kilian, A. Magen, S. Raskhodnikova, R. Rubinfeld, and R. Sami. A sublinear algorithm for weakly approximating edit distance. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing (STOC '03)*, pages 316–324, June 2003.

[7] E. R. Berlekamp. *Algebraic Coding Theory*. McGraw-Hill, Inc., New York, New York, 1968.

[8] M. W. Berry, Z. Drmač, and E. R. Jessup. Matrices, vector spaces, and information retrieval. *SIAM Review*, 41(2):335–362, 1999.

[9] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. http://www-db.stanford.edu/~backrub/google.html.

[10] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th International World Wide Web Conference*, pages 107–117, Brisbane, Australia, 1998. Elsevier Science Publishers B.V. Amsterdam, The Netherlands, The Netherlands.

[11] A. Z. Broder. Some applications of Rabin's fingerprinting method. In R. Capocelli, A. D. Santis, and U. Vaccaro, editors, *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.

[12] A. Z. Broder. On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of Sequences (SEQUENCES '97)*, pages 21–29. IEEE Computer Society, 1998.

[13] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing (STOC '98)*, pages 327–336, 1998.

[14] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *Journal of Computer and Systems Sciences*, 60(3):630–659, 2000.

[15] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. In *Proceedings of the 6th International World Wide Web Conference*, pages 391–404, Santa Clara, California, Apr. 1997.

[16] P. Buneman, S. Khanna, K. Tajima, and W. C. Tan. Archiving scientific data. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, 2002.

[17] T. Burkard. Herodotus: A peer-to-peer web archival system. Master's thesis, Massachusetts Institute of Technology, May 2002.

[18] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236, Apr. 1973.

[19] R. Burns. Differential compression: a generalized solution for binary files. Master's thesis, University of California, Santa Cruz, Dec. 1996.

[20] R. Burns and D. D. E. Long. Efficient distributed back-up with delta compression. In *Proceedings of I/O in Parallel and Distributed Systems (IOPADS '97)*, pages 27–36, San Jose, California, Nov. 1997. ACM Press.

[21] R. Burns and D. D. E. Long. A linear time, constant space differencing algorithm. In *Proceedings of the 16th IEEE International Performance, Computing and Communications Conference (IPCCC '97)*, pages 429–436, Phoenix, Arizona, Feb. 1997. IEEE.

[22] R. Burns and D. D. E. Long. In-place reconstruction of delta compressed files. In *Proceedings of the Seventeenth ACM Symposium on Principles of Distributed Computing (PODC 1998)*, pages 267–275, Puerto Vallarta, Mexico, June 1998. IEEE.

[23] R. Burns, L. Stockmeyer, and D. D. E. Long. Experimentally evaluating in-place delta reconstruction. In *Proceedings of the 19th IEEE Symposium on Mass Storage Systems and Technologies*. IEEE, 2002.

[24] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Systems Research Center, May 1994.

[25] L.-F. Cabrera and D. D. E. Long. Swift: A distributed storage architecture for large objects. In *Digest of Papers, 11th IEEE Symposium on Mass Storage Systems*, pages 123–128, Monterey, California, Oct. 1991. IEEE.

[26] L.-F. Cabrera and D. D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4):405–436, 1991.

[27] California Spatial Information Library. California digital raster graphics. http://gis.ca.gov/casil/gis.ca.gov/drg/.

[28] CAMiLEON Project: Creative Archiving at Michigan & Leeds: Emulating the Old on the New. http://www.si.umich.edu/CAMILEON/.

[29] F. Can. Incremental clustering for dynamic information processing. *ACM Transactions on Information Systems*, 11(2):143–164, Apr. 1993.

[30] A. Cannane and H. E. Williams. A general-purpose compression scheme for large collections. *ACM Transactions on Information Systems*, 20(3):329–355, 2002.

[31] J. L. Carter and M. N. Wegman. Universal classes of hash functions (extended abstract). In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing (STOC '77)*, pages 106–112, New York, New York, 1977. ACM Press.

[32] C. Chan. Don't rule out tape, experts say. http://www.itworld.com/nl/storage_sol/12092002/pf_index.html, Dec. 2002.

[33] M. Charikar, C. Chekuri, T. Feder, and R. Motwani. Incremental clustering and dynamic information retrieval. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing (STOC '97)*, pages 626–635, El Paso, Texas, 1997. ACM Press.

[34] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2), June 1994.

[35] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti, and P. Yianilos. A prototype implementation of archival intermemory. In *Proceedings of the Fourth ACM International Conference on Digital Libraries (DL '99)*, pages 28–37. ACM Press, 1999.

[36] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, , and S. Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, San Francisco, California, Dec. 2004. USENIX Association.

[37] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 285–298, New York, New York, 2002. ACM Press.

[38] A. Crespo and H. Garcia-Molina. Archival storage for digital libraries. In *Proceedings of the Third ACM International Conference on Digital Libraries (DL '98)*, pages 69–78, Pittsburgh, Pennsylvania, June 1998. ACM Press.

[39] J. Dean and M. R. Henziger. Finding related pages in the World Wide Web. In *Proceedings of the 8th International World Wide Web Conference*, Toronto, Ontario, Canada, May 1999.

[40] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *Proceedings of the 2003 USENIX Annual Technical Conference*, San Antonio, Texas, June 2003.

[41] F. Douglis, A. K. Iyengar, and K.-P. Vo. Dynamic suppression of similarity in the web: a case for deployable detection mechanisms. Technical Report RC22514, IBM Research, Thomas J. Watson Research Center, Yorktown Heights, New York, July 2002.

[42] EMC Corporation. EMC Centera: Content Addressed Storage System, Data Sheet. http://www.emc.com/pdf/products/centera/centera_ds.pdf, Apr. 2002.

[43] D. Eppstein. Fast hierarchical clustering and other applications of dynamic closest pairs. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, pages 619–628, Jan. 1998.

[44] Éric Fimbel. Edit distance and Chaitin-Kolmogorov difference. Technical Report ETS-RT-2002-001, École de Technologie Supérieur, Université du Québec, Sept. 2002.

[45] K. Eshghi. Intrinsic references in distributed systems. Technical Report HPL-2002-32, HP Laboratories, Feb. 2002.

[46] D. Fetterly, M. Manasse, M. Najork, and J. Wiener. A large-scale study of the evolution of web pages. In *Proceedings of the 12th International World Wide Web Conference*, pages 669–678. ACM Press, May 2003.

[47] R. A. Finkel, A. Zaslavsky, K. Monostori, and H. Schmidt. Signature extraction for overlap detection in documents. In M. J. Oudshoorn, editor, *Proceedings of the Twenty-Fifth Australasian Computer Science Conference (ACSC2002)*, Melbourne, Australia, 2002. ACS.

[48] A. Finney. The Domesday Project—November 1986. http://www.atsf.co.uk/dottext/domesday.html, Feb. 2004.

[49] Free Software Foundation. The gzip data compression program. http://www.gnu.org/software/gzip/gzip.html, 2000.

[50] J. Gailly and M. Adler. Zlib lossless data compression library. http://www.gzip.org/zlib/.

[51] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, Oct. 2003. ACM Press.

[52] T. Gibson, E. L. Miller, and D. D. E. Long. Long-term file activity and inter-reference patterns. In *Proceedings of the 24th International Conference for the Resource Management and Performance and Performance Evaluation of Enterprise Computing Systems (CMG98)*, pages 976–987, Anaheim, California, Dec. 1998. CMG.

[53] T. J. Gibson. *Long-term UNIX File System Activity and the Efficacy of Automatic File Migration*. PhD thesis, University of Maryland, Baltimore County, May 1998.

[54] A. V. Goldberg and P. N. Yianilos. Towards an archival intermemory. In *Proceedings of IEEE Advances in Digital Libraries, ADL '98*, pages 147–156, Santa Barbara, California, Apr. 1998. IEEE Computer Society.

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

[56] J. Gray, W. Chong, T. Barclay, A. Szalay, and J. Vandenberg. TeraScale SneakerNet: Using inexpensive disks for backup, archiving, and data exchange. Technical Report MS-TR-02-54, Microsoft Research, Advanced Technology Division, Redmond, Washington, May 2002.

[57] J. Gray and P. Shenoy. Rules of thumb in data engineering. In *Proceedings of the 16th International Conference on Data Engineering (ICDE '00)*, pages 3–12, San Diego, California, Mar. 2000. IEEE.

[58] D. R. Hardy and M. F. Schwartz. Essence: A resource discovery system based on semantic file indexing. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 361–374, 1993.

[59] T. H. Haveliwala, A. Gionis, D. Klein, and P. Indyk. Evaluating strategies for similarity search on the web. In *Proceedings of the 11th International World Wide Web Conference*, pages 432–442. ACM Press, May 2002.

[60] V. Henson. An analysis of compare-by-hash. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*. USENIX Association, May 2003.

[61] T. Hey and A. Trefethen. *Grid Computing: Making the Global Infrastructure a Reality*, chapter The Data Deluge: An e-Science Perspective. John Wiley & Sons, Apr. 2003.

[62] Hitachi Global Storage Technologies. Hitachi hard disk drive specification: Deskstar 7K400 3.5 inch Ultra ATA/133 and 3.5 inch Serial ATA hard disk drives, version 1.4, Aug. 2004.

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

[64] G. R. Hjaltason and H. Samet. Incremental similarity search in multimedia databases. Technical Report CS-TR-4199, University of Maryland, College Park, Maryland, Nov. 2000.

[65] J. Hollingsworth and E. L. Miller. Using content-derived names for configuration management. In *Proceedings of the 1997 Symposium on Software Reusability (SSR '97)*, pages 104–109, Boston, Massachusetts, May 1997.

[66] B. Hong, D. Plantenberg, D. D. E. Long, and M. Sivan-Zimet. Duplicate data elimination in a SAN file system. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, College Park, Maryland, Apr. 2004.

[67] M. Hughes. PDF/A as an archive standard. http://www.aiim.org/documents/standards/pdfa2.pdf, Oct. 2002.

[68] J. J. Hunt, K.-P. Vo, and W. F. Tichy. Delta algorithms: an empirical analysis. *ACM Transactions on Software Engineering and Methodology*, 7(2):192–214, 1998.

[69] J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Technical Report CSTR 41, Bell Laboratories, Murray Hill, New Jersey, 1976.

[70] IBM. IBM Tivoli Storage Manager. http://www.tivoli.com/products/solutions/storage/.

[71] IBM. IBM OEM hard disk drive specification for DPTA-3xxxxx 37.5 GB–13.6 GB 3.5-inch hard disk drive with ATA interface, revision (2.1). (Deskstar 34GXP and 37GP hard disk drives), July 1999.

[72] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing (STOC '98)*, pages 604–613. ACM Press, 1998.

[73] ISO Archiving Standards—Overview. http://ssdoo.gsfc.nasa.gov/nost/isoas/overview.html, 2003.

[74] ISO/CD 19005-1: Document management—Electronic document file format for long-term preservation—Part 1: Use of PDF (PDF/A), Oct. 2003.

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

[76] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, Mar. 1987.

[77] R. H. Katz, G. A. Gibson, and D. A. Patterson. Disk system architectures for high performance computing. *IEEE*, 77(12):1842–1858, Dec. 1989.

[78] J. T. Kohl, C. Staelin, and M. Stonebraker. HighLight: Using a log-structured file system for tertiary storage management. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 435–447, San Diego, California, Jan. 1993.

[79] D. Korn, J. MacDonald, J. Mogul, and K. Vo. The VCDIFF generic differencing and compression data format. Request For Comments (RFC) 3284, IETF, June 2002.

[80] D. G. Korn and K.-P. Vo. Engineering a differencing and compression data format. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 219–228, Monterey, California, June 2002. USENIX Association.

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

[82] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th Interational Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM Press, Nov. 2000.

[83] P. Kulkarni, F. Douglis, J. LaVoie, and J. M. Tracey. Redundancy elimination within large collections of files. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 59–72, Boston, Massachusetts, June 2004.

[84] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th Interational Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 84–92. ACM Press, 1996.

[85] S. Levenson and M. Warfel. PDF/A background. `http://www.aiim.org/documents/standards/oct3mel.pdf`, Oct. 2002.

[86] H. Liefke and D. Suciu. XMill: An efficient compressor for XML data. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 153–164, May 2000.

[87] W. Litwin and M.-A. Neimat. High-availability LH* schemes with mirroring. In *Proceedings of the Conference on Cooperative Information Systems*, pages 196–205, Jan. 1996.

[88] W. Litwin, M.-A. Neimat, and D. A. Schneider. LH*—linear hashing for distributed files. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 327–336. ACM Press, 1993.

[89] W. Litwin, M.-A. Neimat, and D. A. Schneider. LH*—a scalable, distributed data structure. *ACM Trans. Database Syst.*, 21(4):480–525, 1996.

[90] W. Litwin and T. Schwarz. Algebraic signatures for scalable distributed data structures. In *Proceedings of the 20th International Conference on Data Engineering (ICDE '04)*, Boston, Massachussetts, Mar. 2004. IEEE.

[91] R. A. Lorie. A project on preservation of digital data. http://www.rlg.org/preserv/diginews/diginews5-3.html, June 2001. Volume 5, Number 3.

[92] R. A. Lorie. Long-term archival of digital information. University of California, Santa Cruz, Storage Systems Research Center Seminar, May 2004.

[93] Q. Lv, M. Charikar, and K. Li. Image similarity search with compact data structures. In *Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management (CIKM '04)*, pages 208–217, New York, New York, 2004. ACM Press.

[94] P. Lyman, H. R. Varian, K. Searingen, P. Charles, N. Good, L. L. Jordan, and J. Pal. How much information? 2003. http://www.sims.berkeley.edu/research/projects/how-much-info-2003/, Oct. 2003.

[95] J. P. MacDonald. Xdelta 1.1.3. http://sourceforge.net/projects/xdelta/.

[96] J. P. MacDonald. File system support for delta compression. Master's thesis, University of California, Berkeley, 2000.

[97] K. Magoutis, S. Addetia, A. Fedorova, M. I. Seltzer, J. S. Chase, A. J. Gallatin, R. Kisley, R. G. Wickremesinghe, and E. Gabber. Structure and performance of the direct access file system. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 1–14, Monterey, California, 2002. USENIX Association.

[98] M. Mahalingam, C. Tang, and Z. Xu. Towards a semantic, deep archival file system. Technical Report HPL-2002-199, HP Laboratories, Palo Alto, California, July 2002.

[99] M. Manasse. Finding similar things quickly in large collections. http://research.microsoft.com/research/sv/PageTurner/similarity.htm.

[100] U. Manber. Finding similar files in a large file system. Technical Report TR 93-33, Department of Computer Science, The University of Arizona, Tucson, Arizona, Oct. 1993.

[101] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. Technical Report TR 93-94, Department of Computer Science, The University of Arizona, Tucson, Arizona, Oct. 1993.

[102] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook for Applied Cryptography*, chapter 2. CRC Press, Oct. 1996.

[103] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook for Applied Cryptography*, chapter 4. CRC Press, Oct. 1996.

[104] J. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta-encoding and data compression for HTTP. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '97)*, Cannes, France, Sept. 1997.

[105] R. Moore, C. Barua, A. Rajasekar, B. Ludaescher, R. Marciano, M. Wan, W. Schroeder, and A. Gupta. Collection-based persistent digital archives—part 1. http://www.dlib.org/dlib/march00/moore/03moore-pt1.html, Mar. 2000.

[106] R. Moore, C. Barua, A. Rajasekar, B. Ludaescher, R. Marciano, M. Wan, W. Schroeder, and A. Gupta. Collection-based persistent digital archives—part 2. http://www.dlib.org/dlib/april00/moore/04moore-pt2.html, Apr. 2000.

[107] R. Moore, J. Lopez, C. Lofton, W. Schroeder, G. Kremenek, and M. Gleicher. Configuring and tuning archival storage systems. In *Proceedings of the 7th NASA Goddard Conference on Mass Storage Systems and Technologies*, Mar. 1999.

[108] T. D. Moreton, I. A. Pratt, and T. L. Harris. Storage, mutability and naming in Pasta. In *Web Engineering and Peer-to-Peer Computing, NETWORKING 2002 Workshops, Pisa, Italy, May 19–24, 2002, Revised Papers*, pages 215–219, 2002.

[109] M. N. Murty and G. Krishna. A computationally efficient technique for data-clustering. *Pattern Recognition*, 12:153–158, 1980.

[110] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 174–187, Lake Louise, Alberta, Canada, Oct. 2001.

[111] M. Nelson and J.-L. Gailly. *The Data Compression Book*. M&T Books, New York, New York, 2nd edition, 1996.

[112] Network Appliance. A new approach for strong reference information—NearLine storage. Technical Report TR 3193, Network Appliance, Inc., Sept. 2002.

[113] NIST FIPS 180-1: Secure Hash Standard, Apr. 1995.

[114] NIST FIPS 180-2: Secure Hash Standard, Aug. 2002.

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

[116] E. J. Otoo. Balanced multidimensional extendible hash tree. In *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 100–113. ACM Press, 1986.

[117] M. Ouksel and P. Scheuermann. Storage mappings for multidimensional linear dynamic hashing. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 90–105. ACM Press, 1983.

[118] Z. Ouyang, N. Memon, T. Suel, and D. Trendafilov. Cluster-based delta compression of a collection of files. In *International Conference on Web Information Systems Engineering (WISE 2002)*, pages 257–266. IEEE, Dec. 2002.

[119] PDF Archive Committee. http://www.aiim.org/pdf_a/.

[120] P. K. Pearson. Fast hashing of variable-length text strings. *Communications of the ACM*, 33(6):677–680, 1990.

[121] C. Percival. Naive differences of executable code. http://www.daemonology.net/bsdiff/.

[122] Perkins Coie. Document retention after Sarbanes-Oxley. http://www.perkinscoie.com/content/ren/updates/corp/093003.htm, July 2002.

[123] Pocket Soft, Inc. RTPatch. http://www.pocketsoft.com/.

[124] C. Policroniades and I. Pratt. Alternatives for detecting redundancy in storage systems data. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 73–86, Boston, Massachusetts, June 2004. USENIX Association.

[125] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 89–101, Monterey, California, Jan. 2002. USENIX Association.

[126] M. O. Rabin. Probabilistic algorithms in finite fields. *SIAM Journal on Computing*, 9(2):273–280, May 1980.

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

[128] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[129] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. 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, San Diego, California, 2001. ACM Press.

[130] C. Reichenberger. Delta storage for arbitrary non-text files. In *Proceedings of the 3rd International Workshop on Software Configuration Management*, pages 144–152. ACM Press, 1991.

[131] C. M. Riggle and S. G. McCarthy. Design of error correction systems for disk drives. *IEEE Transactions on Magnetics*, 34(4):2362–2371, July 1998.

[132] R. Rivest. The MD4 message digest algorithm. Request For Comments (RFC) 1186, IETF, Oct. 1990.

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

[134] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, Feb. 1992.

[135] J. Rothenberg. Ensuring the longevity of digital documents. *Scientific American*, 272(1):42–47, Jan. 1995.

[136] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science; Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12–16, 2001. Proceedings*, 2218:329–350, 2001.

[137] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 188–201, New York, New York, 2001. ACM Press.

[138] Y. Saito and C. Karamanolis. Pangaea: A symbiotic wide-area file system. In *Proceedings of the 2002 ACM SIGOPS European Workshop*. ACM Press, Sept. 2002.

[139] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Dec. 2002.

[140] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 110–123, Dec. 1999.

[141] K. Sayood, editor. *Lossless Compression Handbook*. Academic Press, an imprint of Elsevier Science, 2003.

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

[143] Secretary of State Kevin Shelley, State of California. Local government records management guidelines, June 2004.

[144] M. I. Seltzer and O. Yigit. A new hashing package for UNIX. In *Proceedings of the Winter 1991 USENIX Technical Conference*, pages 173–184, 1991.

[145] J. Seward. http://sources.redhat.com/bzip2/, 2002.

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

[147] N. Shivakumar and H. Garcia-Molina. Building a scalable and accurate copy detection mechanism. In *Proceedings of the First ACM International Conference on Digital Libraries (DL '96)*, Bethesda, Maryland, Mar. 1996.

[148] N. Shivakumar and H. Garcia-Molina. Finding near-replicas of documents on the web. In *Proceedings of The Second International Workshop on the World Wide Web and Databases (WebDB '99)*. LNCS, 1999.

[149] Sleepycat Software. Berkeley DB Database. http://www.sleepycat.com/.

[150] C. A. N. Soules and G. R. Ganger. Connections: Using context to enhance file search. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 119–132, Brighton, United Kingdom, Oct. 2005.

[151] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '00)*, pages 87–95, Stockholm, Sweden, Aug. 2000. ACM Press.

[152] R. Stata, K. Bharat, and F. Maghoul. The term vector database: Fast access to indexing terms for web pages. In *Proceedings of the 9th International World Wide Web Conference*, Amsterdam, Kingdom of the Netherlands, May 2000.

[153] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. 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. ACM Press, 2001.

[154] A. S. Tanenbaum, J. N. Herder, and H. Bos. File size distribution on UNIX systems: then and now. *ACM SIGOPS Operating Systems Review*, 40(1):100–104, 2006.

[155] C. Tang, Z. Xu, and M. Mahalingam. PeerSearch: Efficient information retrieval in peer-to-peer networks. Technical Report HPL-2002-198, HP Laboratories, Palo Alto, California, July 2002.

[156] The Enterprise Storage Group. *Reference Information*: The Next Wave; The Summary of: "A Snapshot Research Study by The Enterprise Storage Group". http://www.enterprisestoragegroup.com/, 2002.

[157] The Enterprise Storage Group. Compliance: The effect on information management and the storage industry. http://www.enterprisestoragegroup.com/, 2003.

[158] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: a scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 224–237. ACM Press, 1997.

[159] W. F. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–321, 1984.

[160] W. F. Tichy. RCS—a system for version control. *Software—Practice and Experience*, 15(7):637–654, July 1985.

[161] M. Tong. *General Hashing*. PhD thesis, University of Auckland, Nov. 1996.

[162] D. Trendafilov, N. Memon, and T. Suel. Zdelta: An efficient delta compression tool. Technical Report TR-CIS-2002-02, Polytechnic University, June 2002.

[163] D. Trendafilov, N. Memon, and T. Suel. Compression file collections with a TSP-based approach. Technical Report TR-CIS-2004-02, Polytechnic University, Apr. 2004.

[164] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, Feb. 1999.

[165] U.S. House. Sarbanes-Oxley Act of 2002. 107th Cong., 2d sess., H.R.3763, Jan. 2002.

[166] X. Wang, D. Feng, X. Lai, and H. Yu. Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint Archive, Report 2004/199, 2004. http://eprint.iacr.org/.

[167] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitiative comparison. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, Massachusetts, Mar. 2002.

[168] H. Weatherspoon, C. Wells, P. R. Eaton, B. Y. Zhao, and J. D. Kubiatowicz. Silverback: A global-scale archival system. Technical Report CSD-01-1139, University of California, Berkeley, Mar. 2000.

[169] G. Wiederhold. *Database Design*. McGraw-Hill, New York, New York, second edition, 1983.

[170] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, second edition, 1999.

[171] T. Wong, C. Wang, and J. Wing. Verifiable secret redistribution for archive systems. pages 94–105, Greenbelt, Maryland, Dec. 2002.

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

[173] L. L. You and C. Karamanolis. Evaluation of efficient archival storage techniques. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 227–232, College Park, Maryland, Apr. 2004.

[174] L. L. You, K. T. Pollack, and D. D. E. Long. Deep Store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, Tokyo, Japan, Apr. 2005. IEEE.

[175] L. Yu and D. J. Rosenkrantz. A linear-time scheme for version reconstruction. *ACM Transactions on Programming Languages and Systems*, 16(3):775–797, 1994.

[176] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: an efficient data clustering method for very large databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 103–114, Montreal, Quebec, Canada, June 1996.

[177] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, May 1977.