RJ 10305 (A0310-017) October 7, 2003
Computer Science

IBM Research Report

Duplicate Management for Reference Data

Timothy E. Denehy
Computer Sciences Department
University of Wisconsin
Madison, WI 53706

Windsor W. Hsu
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

Duplicate Management for Reference Data[*]

Timothy E. Denehy
Computer Sciences Department
University of Wisconsin
Madison, WI 53706
tedenehy@cs.wisc.edu

Windsor W. Hsu
Computer Science Storage Systems Department
IBM Almaden Research Center
San Jose, CA 95120
windsor@almaden.ibm.com

January 28, 2004

Abstract

Recent studies show that reference or fixed content data accounts for more than half of all newly created digital data, and is growing rapidly. Reference data is characterized by enormous quantities of largely similar data and very long retention periods. Their secure retention and eventual destruction are increasingly regulated by government agencies as more and more critical data are stored electronically and are vulnerable to unauthorized destruction and tampering. In this paper, we describe a storage system optimized for reference data. The system manages unique chunks of data to reliably and efficiently store large amounts of similar data, and to allow selected data to be efficiently shredded. We discuss ways to detect duplicate data, describing a sliding blocking method that greatly outperforms other methods. We also present practical ways to organize the metadata for the unique chunks, allowing most of it to be kept on disk and to be effectively prefetched when needed. Since electronic mail (email) is an important storage-intensive instance of reference data and is currently the intense focus of regulatory bodies, we use email as a sample application and analyze its storage characteristics in detail. We find that more than 30% of the blocks in an email data set are duplicates and that a duplicate block is most likely to occur within a few days of its previous occurrence. Our analysis further indicates that the effects of duplicate block elimination and compression techniques such as block gzip seem to be relatively independent so that they can be combined to achieve additive results.

[*]Work conducted at IBM Almaden Research Center, San Jose, CA as a summer intern project in 2003.

1 Introduction

Reference data, also known as fixed content or archive data, is data that does not change or that needs to be frozen at a point in time. Examples of reference data include electronic mail (email), instant messages, insurance claims processing, drug development logs, bio-informatics, digitized information such as check images, medical images, surveillance video and x-rays. Reference data is characterized by enormous quantities of largely similar data (multiple petabytes) and very long retention periods, up to and sometimes even beyond 25 years. According to the research group, Enterprise Storage Group (ESG), reference data is growing at 68% annually, and is expected to be a larger market opportunity than transaction data by 2007 [2]. Another study found that 75% of new digital data is fixed content [13].

As critical data is increasingly stored in electronic form, a growing subset of reference data is subject to regulations governing their long-term retention and availability. Recent high-profiled accountability issues at large public companies have further caused regulatory bodies such as the Securities and Exchange Commission (SEC) to tighten their regulations. For instance, SEC's new Rule 17a-4, which went into effect in May 2003, specifies storage requirements for email, attachments, memos and instant messaging as well as routine phone conversations. There are generally four key storage requirements in such rules:

1. Store data reliably on non-erasable, non-rewriteable media. The reliability is typically ensured by keeping multiple copies of the data, which significantly increases the cost of the system. The non-erasable and non-rewriteable media used to mean optical media but has recently been expanded to include WORM-tape and disk-based storage with a content-addressable interface [10]. Although reference data is sometimes described as write once read rarely (or read-never), there are many environments where read performance does matter. Moreover, when discovery requests for data arrive, the data needs to be readily accessible, in random and sequential modes depending on the application.

2. Enforce retention policy for data, making sure that data is not deleted before its retention period has expired. As data life cycle lengthens, total cost of ownership becomes a key consideration.

3. Shred data after its retention period has expired so that the data cannot be recovered or discovered by data forensics. This is typically accomplished by overwriting the data many times (more than eight in some cases).

4. Maintain an audit trail of who created the data, who accessed it and when.

With the increasing value of electronic data to companies and organizations, and the threat of unfavorable data discovery during litigation, these storage requirements are increasingly attractive even for non-regulated data.

In this paper, we outline a storage system that is geared towards managing the rapidly increasing amount of reference data. The system manages unique chunks of data to reliably and efficiently store large amounts of similar data, and to allow selected data to be effectively shredded. A key part of the system is a sliding blocking method for detecting duplicate data that outperforms previous methods and that results in predominantly fixed-size chunks that are much easier to manage. In addition, we present an age-based approach to organizing the required metadata, which allows most of it to be kept on disk and to be effectively prefetched when needed.

We use email as a sample workload, analyzing its storage characteristics in detail, because email is an important and increasingly storage-intensive instance of reference data. A recent survey of 146 IT managers by Osterman Research [1] shows that email storage requirements have increased by 36% in the last 12 months. Moreover, email is the intense focus of regulatory bodies and millions of dollars in fines have recently been levied by SEC on Wall Street firms for not properly archiving their email. We find that more than 32% of the blocks in an email data set are duplicates. Our analysis also indicates that attachments constitute as much as 80% of the email data and that to detect duplicate attachments, it is sufficient to compare attachments with the same name. However, many attachments that are not exact duplicates of another do share many common blocks. We also discover that a duplicate block is most likely to occur within a few days of its previous occurrence.

The rest of this paper is organized as follows. In the next section, we present our system for managing unique data. In Section 3, we discuss various techniques for detecting duplicate data. A comparative analysis of the effectiveness of these techniques on real-world data is presented in Section 4. We characterize the email workload in Section 5 and discuss related work in Section 6. Section 7 concludes this paper.

2 Management of Unique Data

Storage systems today manage data storage on the basis of the containers that store the data (*e.g.,* blocks). The exact same data could be in multiple blocks in a system but the system would be unaware of the fact, and would be unable to exploit the fact for cost savings, performance improvement, reliability enhancement *etc.* For example, when a copy of data is lost due to hardware problems (*e.g.,* media failure, which is especially a problem when low-cost desktop drives are used), the system may not be able to repair that copy even though there are many other identical copies of the data located elsewhere in the system. It simply has no idea exactly how many copies there are or where all the copies are located. Solutions today rely on maintaining redundancy of all the blocks, which is very costly, rather than controlled redundancy of the actual data, which is what is really needed.

In this paper, we propose a system that manages data storage based on the information content across the blocks. The system figures out which are the unique blocks of data, and stores some number of copies of such blocks to achieve, at low cost, the desired levels of performance and fault-tolerance. As we describe in the next section, we can additionally perform compression on the data to further reduce storage usage. In addition, methods such as that presented in [5] can be used to delta-compress similar but non-identical blocks. We note, however, that delta compression results in small variable-sized chunks of data, which are difficult to manage and which adds overhead to the system. Moreover, delta compression requires keeping a reference copy of the data against which to perform the deltas, resulting in a complicated arrangement when data has to be shredded.

2.1 Major Components

The major components of the system are as follows:

1. *Object table* (OT). The data items in reference data are commonly referred to as records or objects. The OT maps an object to the locations (addresses) of the storage (disk) blocks that belong to that object.

2. *Free space map* (FSM). The FSM keeps track of the unallocated storage blocks, *i.e.*, the storage blocks that do not belong to any object.

3. Occurrence count analyzer. The occurrence count of a block of data is the number of times that block of data has been logically stored. The system determines the occurrence count of a block of data by hashing the data into a hash table called the *block contents map* (BCM). A cryptographic hash function (*e.g.*, SHA-1 [9]) can be used to reduce the chances for collisions. Each entry in the BCM corresponds to a unique block of data, and contains the occurrence count for that block of data, the location and reference count of the storage blocks where it is stored, and the IDs of the objects that contain it. Note that an object may contain the same block of data multiple times so the occurrence count can be larger than the number of objects containing that block of data.

   In order to support the delete and read error recovery functions, the system also maintains a reverse BCM (rBCM), which simply maps the address of a storage block to its BCM entry. The data contained in the BCM and rBCM are part of the metadata of the object system. Ways to keep the metadata consistent and to bring them back into consistency after some failure are well-known and include synchronous write of metadata to stable storage, logging, soft updates [6], *etc.*

4. Redundancy management. For each unique block of data (as opposed to block of storage), the system decides how many copies of the data to maintain based on a combination of the occurrence count of the data, performance and reliability settings, type of object, *etc.* The system can also use the number of objects containing the data, instead of or in addition to the occurrence count, to determine the number of copies to maintain. Note that storage blocks may have correlated failures. For instance, when a disk crashes, all the storage blocks on the disk fail together. It is preferred that the underlying physical storage system exports its failure boundaries so that the copies of a given block of data can be placed in independent failure domains.

   A mapping or virtualization layer is needed to locate where the multiple copies of a block of data are. This mapping can be inserted into the I/O path just above the physical storage system in which case the system can decide dynamically which copy of a block of data to use. In the system described above, the mapping is integrated into the BCM and kept off the I/O path. It is used only when updating the OT and therefore has no impact on read performance.

5. Schedule control. Determining the occurrence count and managing the data redundancy can be a costly process. The process of determining the occurrence count, and managing the desired number of copies of each unique block of data can be performed inline or deferred until the next convenient time. In the latter case, the system handles the file write operation as in a traditional file system, and reads the block back later for processing.

   In such a system, the storage blocks are categorized into three pools - the common pool, private pool, and unallocated pool. The common pool contains storage blocks that may belong to more than one object or to multiple locations in an object. The private pool contains storage blocks that belong to exactly one object. The free pool contains storage blocks that are not allocated. During an object delete operation, a block in the private pool can be returned to the free pool right away. A block in the common pool can be returned to the free pool only if the block is not used in another object.

2.2 Operations

When a block of data is written to the object system, the system first determines how many copies of that data already exist logically in the file system. Based on this occurrence count, and the performance and reliability settings, the system then decides how many copies of that data it should maintain. If it currently has fewer than the desired number of copies of the data, the system proceeds to create a new copy of the data. If the system already has the desired number of copies of the data, it selects one of the existing copies of the data. In either case, the OT is updated to include a reference to the created/selected copy of the data.

On an object delete, the system goes through the OT to find all the storage blocks belonging to the object to be deleted. The reference count for each of these storage blocks is decremented by the number of references to that block by the current object. When the count reaches zero, that storage block is deallocated and the FSM is updated accordingly. The OT entry corresponding to that file is then removed. As a final step, the system reevaluates the desired number of copies for each block of data that has been affected, rebalancing the number of references to each copy of the block if necessary.

When a read error is encountered for a storage block, the system consults the rBCM to locate the BCM entry of the data stored in the error block. If the BCM entry indicates that there is another copy of the data somewhere else in the storage system, the system reads one of the other copies to satisfy the current read request. It then marks the error block as bad in the FSM, and proceeds to repair the system by going through the OT entries for the files with references to the error copy and changing these references to point to one of the other copies. As a final step, it reevaluates the desired number of copies for that block of data, creating a new copy and rebalancing the references if necessary.

2.3 Metadata Organization

In addition to the metadata maintained by a standard object system, the proposed system maintains a BCM entry for each unique block of data and a rBCM entry for each block of storage. Given the large volume and long retention period of reference data, such metadata will be large and will need to be paged. The desirable property of hashes, however, is that there is no locality in the hash values. In other words, if the BCM entries are stored according to their the hash values, each lookup of the BCM will incur a random I/O.

In many cases, however, similar objects tend to differ only in a handful of blocks so that long sequences of hash values tend to recur. This suggests that we should index BCM entries based on their hash values and cluster them based on when they are created or last updated. With such an approach, we can effectively sequential prefetch subsequent

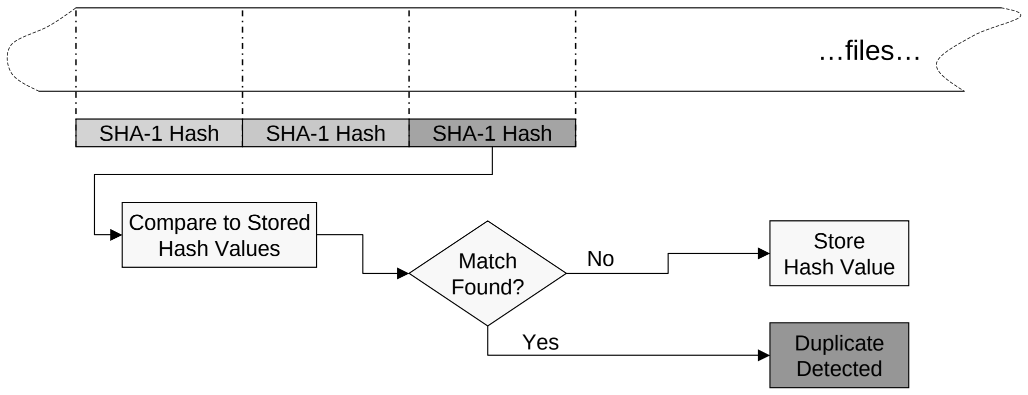

Figure 1: **Simple Blocking Method.** *Each fixed-size file block is hashed. Resulting hash value is compared to prior values to find duplicate blocks.*

BCM entries after an initial random I/O for the first BCM lookup.

Our second insight is that data usage tends to exhibit temporal locality so that any duplicate blocks are likely to occur close together in time. In other words, if we keep the BCM and rBCM entries only for a short duration (*e.g.*, 2 weeks) after they are created or last updated, we can drastically reduce the size of the tables and yet reap most of the benefits discussed here. The BCM and rBCM entries that are aged out can be archived and brought back when needed to recover from a read error. Later in this paper, we will use real email data to validate this insight.

2.4 Prototype Implementation

We have started to implement a simple prototype of the system as a stackable file system in Linux by extending the wrapfs template [14]. In the prototype implementation, an object is stored as one or more files in the underlying file system. Each of these files contain a subsequence of the blocks belonging to an object, and the file is the unit of sharing among different objects. Since similar objects tend to differ only in a few blocks, most of the files are large so that the file overhead is relatively small.

3 Duplicate Detection Methods

In this section we present three methods for detecting duplicate data in a set of objects: simple blocking, content chunking, and sliding blocking.

3.1 Simple Blocking

The simple blocking method illustrated in Figure 1 divides each object into non-overlapping fixed-sized blocks. The SHA-1 hash of each block is computed and stored in a table. As subsequent blocks are examined, their hash values are computed and compared to the stored values in order to detect duplicates. Extra storage is incurred for the table of block hash values and the disk addresses of the associated blocks.

The major problem with this method is that it may ignore the duplication in two objects that differ by only a small number of bytes. For example, imagine that an existing object is copied and a small number of bytes are inserted at the beginning of the new object. The inserted bytes cause a shift in the object contents, and all of the block boundaries following the insertion will be affected. An analogous situation occurs if bytes are deleted from the object. None of the blocks in the new object will exactly match those in the source object, so the remaining similarities between the objects will not be detected.

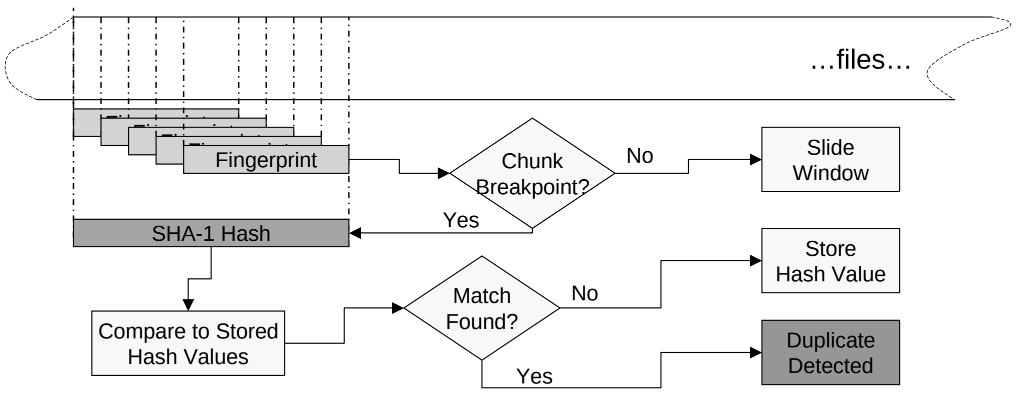

Figure 2: **Content Chunking Method.** *Rabin's fingerprint is computed over a sliding window. Chunks are terminated when the fingerprint takes on a certain predetermined breakpoint value. Chunks are hashed to find duplicate data.*

3.2 Content Chunking

In order to overcome this problem, the low-bandwidth network file system [8] divided objects (files) into variable-sized chunks based on object content. We will refer to this as the content chunking method (Figure 2).

Objects are divided into chunks based on their content using Rabin fingerprints [3, 11] because they are efficient to compute over a sliding window. A 48 byte sliding window is used to calculate the fingerprint for each overlapping 48 byte segment of the object. The fingerprint window moves along the object identifying chunk breakpoints when the low-order bits of the fingerprint match a constant pre-determined breakpoint value. The portions of the object between these breakpoint values are classified as chunks, and the expected chunk size is determined by the number of bits used to compare against the breakpoint value. In order to avoid pathological cases, LBFS suggests setting limits on the sizes of chunks. We use minimum and maximum chunk sizes of 512 bytes and 64 kilobytes, respectively. To calculate the fingerprint, we use a randomly chosen irreducible polynomial of degree 384.

The content chunking method is resilient to the byte insertion problem described above because the chunk boundaries are based on object content instead of length. A small number of bytes inserted into or deleted from a object may or may not affect a chunk breakpoint region. If the change occurs outside of a 48 byte breakpoint region, the chunk boundaries may remain intact, or one chunk may be split into two if the changed bytes create a new breakpoint. If the change occurs within a 48 byte breakpoint region, two chunks may be merged into one if the breakpoint is destroyed, or the boundary between two chunks may move if a new breakpoint is introduced. Most importantly, in each of these cases, inserting or deleting a small number of bytes affects only one or two object chunks. The remaining chunks of the object are unaffected. This allows the content chunking method to detect much more duplication between objects that differ by a small number of bytes.

As an object is divided into chunks, the SHA-1 hash of each chunk is computed and compared to the hash values of previously seen chunks to detect duplicates. Again, extra storage is needed to store the chunk sizes, hash values, and disk addresses.

3.3 Sliding Blocking

The content chunking method solves the insertion problem but introduces the problem of storing variable-sized chunks. For a storage system, it is much less complex and more efficient to store fixed-sized blocks. To this end, we present the sliding blocking method to combine the benefits of the previous approaches and solve the insertion problem while using a fixed block size.

The sliding blocking method (Figure 3) uses the rsync [12] checksum and a block-sized sliding window to calculate the checksum of every over-

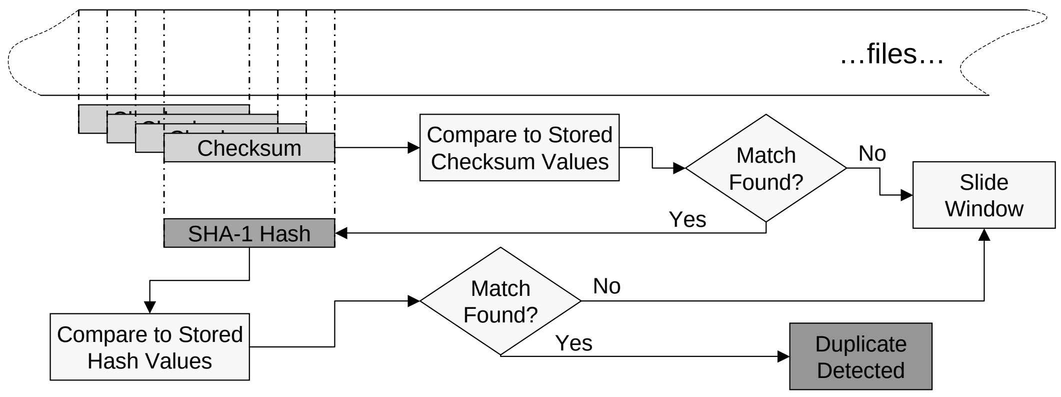

Figure 3: **Sliding Blocking Method.** *A sliding window is used to checksum blocks. A checksum match with prior values leads to a block hash. Hash value is compared to previous values to find duplicate data.*

lapping block-sized segment of an object. The rsync checksum is fast and efficient to compute over a sliding window. The checksum for each block-sized segment is compared to previously stored values. If a match is found, the more expensive SHA-1 hash of the block is calculated and compared to stored hash values to detect duplicates.

If a duplicate is found, it is recorded and the sliding window is moved past the duplicate block to continue the process. Additionally, the fragment of the object between the end of the previous block and the newly detected duplicate must be recorded and stored. When a match for the checksum or the hash value is not found, the sliding window is advanced and the process continues. If the sliding window moves past a full block-sized segment without matching it to a known block, the checksum and SHA-1 hash are computed for that block and stored in their respective tables for comparison against the values for future blocks. This method requires extra storage for both the tables of checksums and hash values as well as the disk addresses of the associated blocks.

The sliding blocking method solves the insertion problem by examining every block-sized segment of the object. If a small number of bytes are inserted into an object, only the surrounding block will change. The next block following the changes will still be identified and matched by the algorithm, and an intervening fragment (equal in length to the inserted bytes) will be created. Similarly, deleting a number of bytes will create a fragment sized by the difference of the block size and the deleted region, but the blocks after the affected region will still be matched by the algorithm.

3.4 Gzip

Standard compression algorithms such as LZ77 [15] used by gzip [4] also detect duplicate data, but typically on a more local scope and at a finer granularity. For example, gzip compresses data by searching for repeated strings in a 32 KB sliding window. When a repeat is found, it is replaced by two numbers: a distance pointer to the previous occurrence in the window and the length of the repeated string. The remaining literals and the pointers are further compressed using a Huffman tree encoding.

Unfortunately, the use of such compression means increased access times because we cannot access the data in an object without decompressing the object from its beginning. In addition, any data loss in an object would render the rest of the object useless. An alternative is to separately compress the blocks of an object. Such block gzip would limit the effect of any data loss and allow direct access to the data in an object but would reduce the compression ratio.

In the next section, we will examine the use of gzip in combination with our other methods in search of space savings over any single method. For instance, gzip could be used to compress the unique blocks that result from duplicate detection. On the other hand, performing gzip or block gzip

Data Set	Description	Files	Size
Email-1	Industry Research Email - Lotus Notes format	127	16.51 GB
Email-2	University Research Email - mbox format	5,819	7.97 GB
Email-3	Computer Support Request Email - mh format	322,666	4.37 GB
Email-4	Industry Research Email - Lotus Notes format	1	0.61 GB
Development	University Research and Development	399,037	15.36 GB
Users	Industry User Data	185,722	6.47 GB

Table 1: **Data Sets.** *This table summarizes the characteristics of each data set used to compare the duplicate detection methods.*

before duplicate detection will eliminate a lot of the duplicates because any changes in an object would percolate down to the end of the compressed object. In this case, a solution is to gzip the object in chunks determined by the content chunking method.

4 Comparison of Methods

We used several real-world data sets to evaluate the different methods for duplicate detection. Each of the data sets is summarized in Table 1. The Email-1 data set consists of email from 127 researchers, managers, and supporting staff at an IBM research laboratory. The emails are stored in the Lotus Notes NSF database format. Email-2 contains email data from 214 members of a university computer science department. The email is stored in the mbox format, which uses one large file to store the contents of each email folder.

The Email-3 data set consists of emails from the problem reporting and request system of a large computer support department. The request system uses the mh format, so each email message is stored in its own file. The Email-4 data set contains the email of a single industry researcher stored in the NSF format. It will be used later to analyze characteristics specific to email messages. Finally, to compare our methods on other forms of data, we used the Development data set and the Users data set. The Development data set consists of source code repositories and sandboxes, object files, experimental data, and reports from a university computer science research group. The Users data set

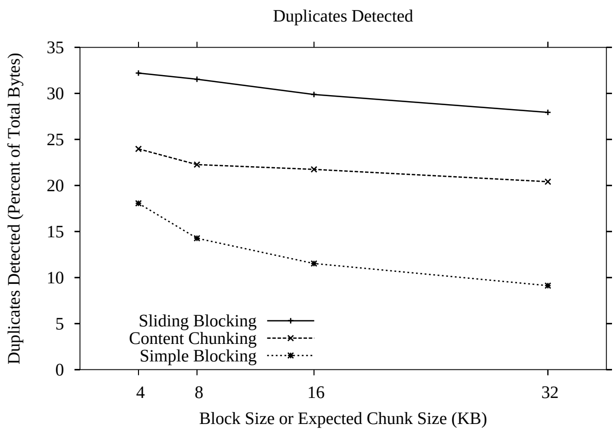

Figure 4: **Duplicates Detected, Email-1.** *This graph shows the number of duplicate bytes detected as a percentage of the total number of bytes using each method on the Email-1 data set. The block size for the sliding blocking and simple blocking methods and the expected chunk size for the content chunking method are varied along the x-axis.*

contains the data of 33 users stored in one partition of a shared storage system within IBM.

First, we will concentrate in detail on the results from the Email-1 data set. Figure 4 shows the amount of duplicate bytes detected by each method as a percentage of the total number of bytes in the data set. The block size for the simple and sliding blocking methods and the expected chunk size for the content chunking method are varied along the x-axis. As expected, the number of duplicates detected decreases slightly as the block size increases because of the coarser granularity. The simple blocking method consistently finds the fewest number of duplicate bytes and the sliding

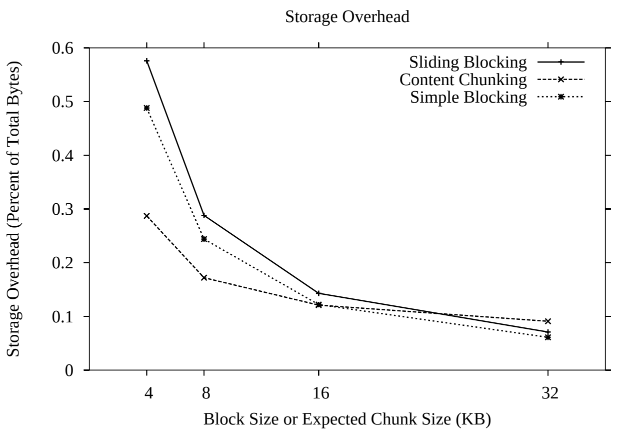

Figure 5: **Storage overhead, Email-1.** *This graph shows the extra storage required by each method as a percentage of the total number of bytes for the Email-1 data set. The block size is varied along the x-axis.*

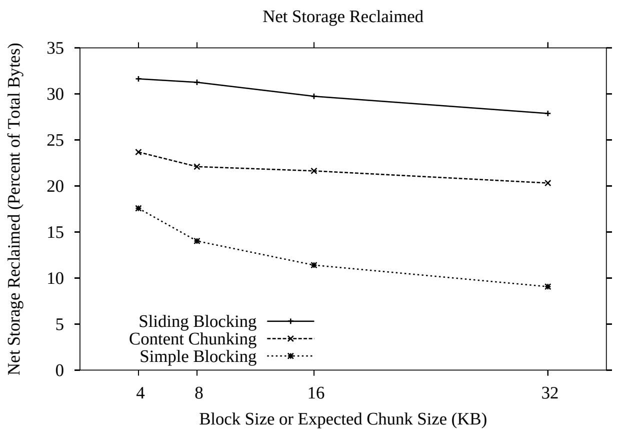

Figure 6: **Net Storage Reclaimed, Email-1.** *This graph shows the net storage reclaimed (the difference between the duplicates detected and the storage overhead) by each method as a percentage of the total number of bytes for the Email-1 data set. The block size is varied along the x-axis.*

blocking method always finds the most duplicates, which in this case is more than 32%.

Note that Lotus Notes already performs compression on some of the attachments using either Huffman tree encoding or LZ compression. As discussed earlier, such compression greatly reduces the amount of duplicates in the data. Furthermore, our results reflect active and expensive management of email by the users. In a regulatory-compliant system where all email is retained for several years, we would expect the amount of duplicate data to be even higher. There is clearly a lot of potential for cost and reliability improvements by managing the unique data.

Figure 5 shows the amount of extra storage needed as a percentage of the total number of bytes in the data set for tracking the extra information required by each method. In all cases the amount of overhead is very small at less than one percent. Figure 6 combines the data from the previous two figures to present the total amount of storage that can be reclaimed by eliminating duplicates with each method. The sliding blocking method remains the best overall. Additionally, the amount of storage that can be reclaimed is substantial, up to 31% in the best case. Note that such savings are especially significant given the extended life of reference data, the need to keep multiple copies of such data to ensure reliable storage, and the fact that while raw storage is increasingly inexpensive, long-term managed storage is not.

The content chunking method performed worse than expected, and we believe this is due to the way in which the data is divided into chunks. Because the data being analyzed is real and not random, the resulting chunk size distributions do not meet expectations. Figure 7 shows the distribution of chunk sizes for the content chunking method with an expected chunk size of 4 KB. The average chunk size turns out to be 6,790 bytes, much larger than the expected value. Furthermore, 64 KB chunks make up 3.4% of all chunks and contain 33.5% of the bytes in the data set. These 64 KB chunks are artificially created due to the maximum chunk size limit. Since they do not end in a chunk breakpoint, they are unlikely to be matched to another large chunk in the data set, and the amount of duplicates that can be detected is reduced. Similar chunk size distributions were observed in our other data sets. Because real data is seldom random, such chunk size distributions will tend to occur in practice, meaning that the content chunking method is unlikely to perform very well in real life.

Each of the duplicate detection methods was also run on the other data sets with a block size or expected chunk size of 4 KB. Figure 8 shows

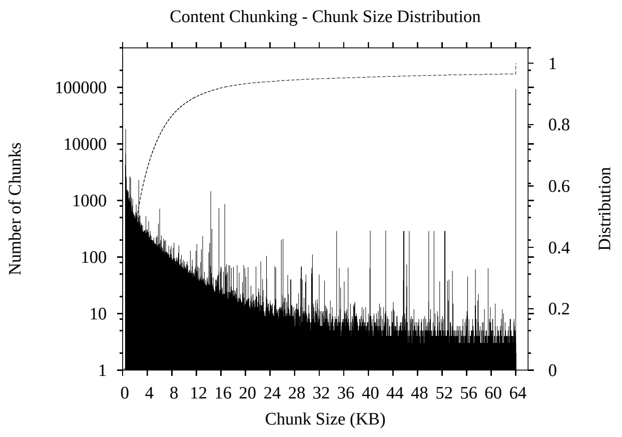

Figure 7: **Chunk Size Distribution, Email-1.** *This graph shows the distribution of chunk sizes for the content chunking method over the Email-1 data set using an expected chunk size of 4 KB. The left y-axis corresponds to the histogram of chunk sizes, and the right y-axis and dotted line show the cumulative distribution of chunk sizes.*

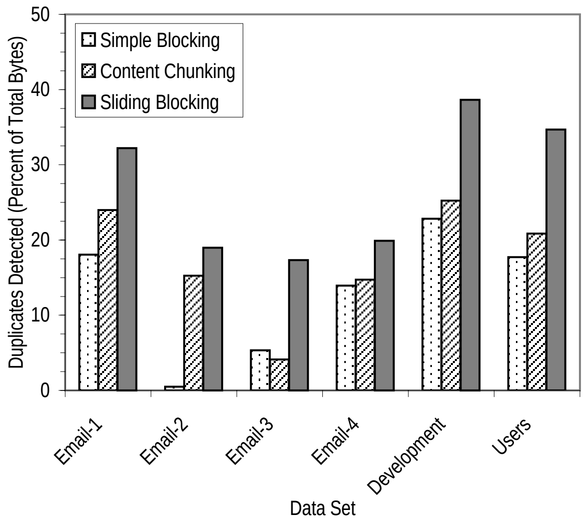

Figure 8: **Duplicates Detected, all Data Sets.** *This graph shows the number of duplicate bytes detected as a percentage of the total number of bytes for each of the methods over the six data sets using a 4 KB block size.*

the percent of duplicate bytes detected by the various techniques. In each case, the sliding blocking method detects the most duplicates, strengthening our previous results for the Email-1 data set alone. While the sliding blocking method finds 18% duplication in the Email-2 data set and 17% in Email-3, the amounts are much lower than the 32% identified in the Email-1 data set. These differences are likely due to the different contents of the data sets, and possibly the more generous mailbox limits in an industry setting. They could also be due to the different storage formats. For example, in the mh format, storing each email in a separate file may increase the number of small file fragments and therefore decrease the chances of finding matching blocks.

For the Email-1, Email-2, and Development data sets, Figure 9 shows the differences in the number of duplicates detected when all of the data is considered together (the default) and when each user's data is examined in isolation. For each data set, the duplicates within each user's data accounts for most (60-80%) of the duplicates detected in the data set. Nevertheless, the combined results are significantly greater, indicating a sizeable amount of sharing between users. This finding suggests that duplicate management could achieve higher cost savings and reliability enhancement when data from multiple users are consolidated onto the same server or same cluster of servers. It could also have implications when assigning users to servers so as to increase local sharing.

We next consider the additional use of gzip compression on the unique blocks detected by the various methods. Figure 10 summarizes the results. The figure shows the results for only two of the data sets because we no longer have access to the other data sets to gather this result. As discussed earlier, the use of gzip to compress objects means increased access times because the data in an object cannot be accessed without first decompressing the object from its beginning. In addition, any data loss in an object would render the rest of the object useless. An alternative is to separately compress the blocks of an object, a method we term block gzip.

Our results show that block gzip is able to achieve greater size reduction than duplicate block elimination for our data sets. However, in a regulatory-compliant system where all the email belonging to many users are retained for several years, we would expect duplicate block elimination to perform better. Moreover, block gzip per-

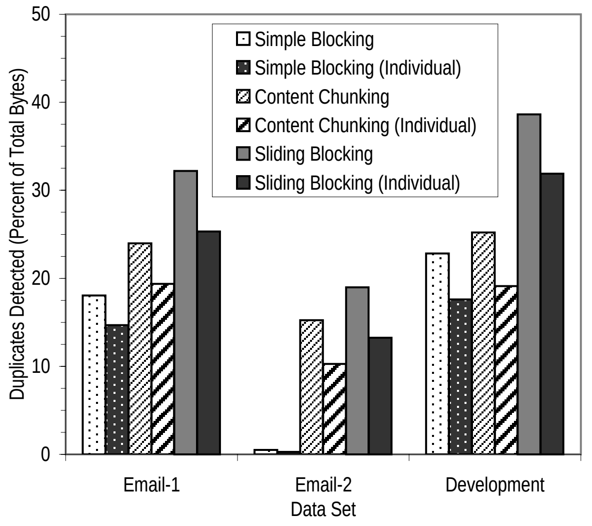

Figure 9: **Duplicates Detected on an Individual Basis.** *This graph shows the number of duplicate bytes detected when all the data was considered in combination (the default) and when each user's data was analyzed for duplicates individually. A 4 KB block size is used.*

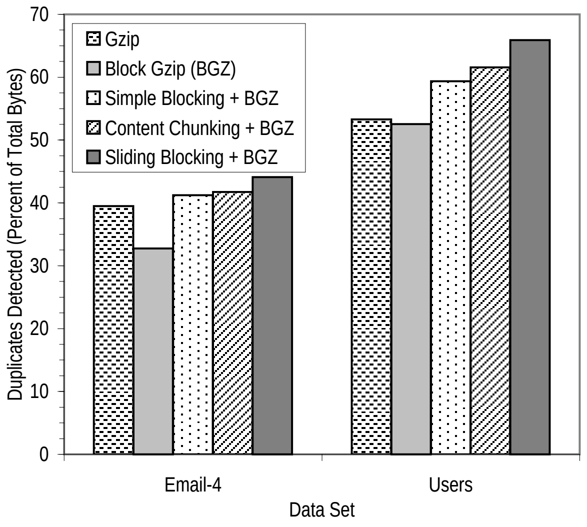

Figure 10: **Duplicates Detected with Additional Use of Block Gzip.** *This graph shows the number of duplicate bytes detected when the unique blocks resulting from each of the methods are compressed with block gzip. A 4 KB block size is used.*

formed after duplicate blocks have been eliminated significantly outperforms block gzip on its own. In fact, the effects of block gzip and duplicate block elimination seem to be relatively independent so that their net effect is additive. Specifically, suppose that duplicate block elimination and block gzip each acting alone reduces the data set by $x\%$ and $y\%$ respectively. We find that when block gzip is performed on the unique blocks remaining after duplicate block elimination, the overall reduction in data size is $x + (1 - \frac{x}{100}) \times y\ \%$.

Table 2 lists the processing rates for our unoptimized implementations of each of the duplicate detection methods. Each technique was used to process in-memory data sets using an Intel Xeon 2.0 GHz processor. The data sets processed include a synthetic 100 MB file with 25% duplicate data aligned on block boundaries (4 KB blocks were used), and the Email-4 data set. The simple blocking method is the fastest and is suitable for on-line processing of file system traffic. The processing rates of the content chunking and sliding blocking methods, at a few megabytes per second, are about an order of magnitude slower. At such rates, these methods are likely to be better suited to off-line processing. We note, however, that we have yet to optimize our implementations of the methods or consider hardware acceleration.

Processing Rate (MB/s)		
Method	Synthetic Data Set	Email-4
Simple Blocking	45.5	45.6
Content Chunking	1.5	1.4
Sliding Blocking	4.5	1.2

Table 2: **Processing Rates.** *This table lists the processing rates for our unoptimized implementations of each duplicate detection method.*

5 Email Characteristics

As discussed earlier, email is an important reference data workload. In this section, we analyze its characteristics in detail using the Email-4 data set. In Table 3, we present some simple statistics of the email. Note that email comes in a wide range of sizes. On average, the size of an email including attachment is 40 KB. Most of the email has only a couple of recipients, and most have no attachments.

	Avg.	Std. Dev.	Min.	Max.	90%-tile
Size of Email (B)	40273	291317	215	10602200	24631
Size of Body (B)	6205	51918	0	2070940	6472
Size of Attachment (B)	233378	626588	0	7368260	539712
# To	1.91	7.66	0	501	2
# CC	0.418	2.78	0	114	1
# BCC	0.00994	0.178	0	18	0
# Attachments	0.139	0.633	0	28	0

Table 3: **Characteristics of Email.** *The column marked 90%-tile contains the 90th percentile of the various measures.*

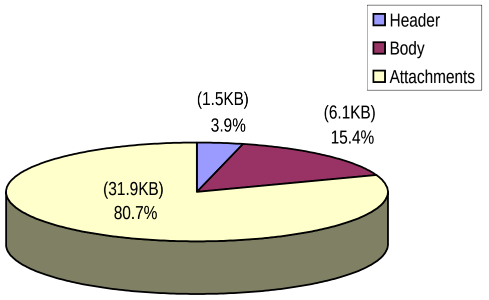

Figure 11: **Relative Sizes of Email Components.** *This chart shows the breakdown of the space occupied by the Email-4 data set. The numbers in parentheses indicate the space occupied by that component averaged over each email.*

Observe from Figure 11 that the attachments form by far the largest component of the email data set, constituting more than 80% of the total data size. The email bodies are also significant, representing about 15% of the total data size. The email headers, which include the sender, recipients, dates, subject and delivery information, represent less than 4% of the data. Such a breakdown suggests that in optimizing the storage of email, we should focus primarily on the attachments and, to a lesser extent, on the email bodies.

In Table 4, we explore the amount of duplicate data in the various email components. We find that comparing attachments with the same name is sufficient to detect most of the duplicate attachments. However, duplicate attachments constitute only about 13% of the space taken up by the attachments. The sliding blocking method, on the other hand, detects 33% duplicate data in the attachments. This suggests that although most of the attachments are not identical, they do share common blocks.

Earlier in the paper, we advocate managing the BCM and rBCM metadata based on its age because any duplicate blocks are likely to occur close together in time. In Figure 12, we empirically demonstrate this behavior. Observe that a duplicate block is most likely to occur within a few days of its previous occurrence. Observe further that the plot using age since first occurrence is significantly below that using age since last occurrence. This suggests that aging the BCM and rBCM entries based on when they were last updated would be more effective than aging them based on when they were first created.

6 Related Work

The rsync algorithm [12] uses duplicate detection to efficiently synchronize a remote copy of a file. The receiver divides its out-of-date version of the file into fixed-sized blocks and sends two checksums of each block to the sender. The sender exams its up-to-date version of the file, calculating the same checksums for every overlapping block-sized string in the file. After detecting duplicate blocks based on the two checksums, the sender transmits data and instructions to the receiver to efficiently update its version of the file.

The low-bandwidth network file system [8] introduced the content chunking method to help reduce the amount of network bandwidth used in a distributed file system. Instead of transferring en-

	Object					
	Email Body	Attachment				Name of Attachment
Duplicate Objects (%)	34.3	32.6				45.3
Detection Method	Compare whole body	Compare whole attachments	Compare attachments with same name	Compare blocks in attachments		-
Duplicate Blocks (%)	18.6	13.2	12.8	32.9		-

Table 4: **Duplicate Percentage by Components.** *The data show that 34% of email bodies are duplicates. Most of the duplicate email bodies are small so that they constitute only 19% of the space occupied by the email bodies. About 33% of the attachments are duplicates but again most of these are small so that they represent only 13% of the space occupied by the attachments. The sliding blocking method, on the other hand, detects 33% duplicate data in the attachments, suggesting that many attachments are similar but not identical. 45% of the attachments have duplicate names. Comparing attachments with the same name is sufficient to discover most of the duplicate attachments.*

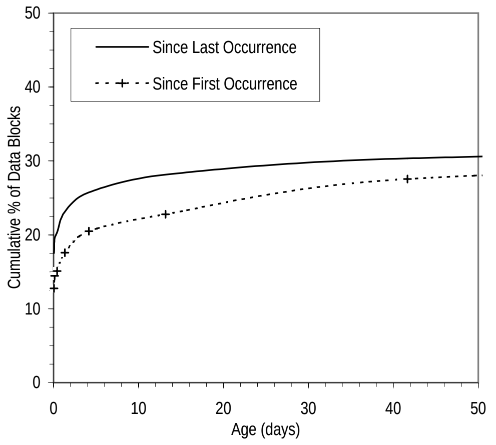

Figure 12: **Age Distribution.** *This graph shows the distribution of the age of a data block belonging to the attachments, where the age is defined as the elapsed time since the data block was last or first encountered.*

tire files across the network, LBFS first divides the file into chunks and then transfers the SHA-1 hash of each chunk to the recipient. The recipient compares the hash values it receives to the hash values of the chunks it is storing in its cache or on its local disk. If matches are found, the amount of data that needs to be transmitted between the sender and receiver is reduced. This scheme has also been adopted by Pasta [7] to reduce duplication in a peer-to-peer storage system.

7 Conclusion

In this paper, we present a storage system for managing the rapidly increasing amount of reference data. The system manages unique chunks of data to reliably and efficiently store large amounts of similar data, and to allow selected data to be effectively shredded. A key part of the system is a sliding blocking method for detecting duplicate data that outperforms previous methods and that results in predominantly fixed-size chunks that are much easier to manage. In addition, we present an age-based approach to organizing the required metadata, which allows most of it to be kept on disk and to be effectively prefetched when needed.

Using real email data as a sample workload, we find that more than 32% of the data are duplicates. In a regulatory-compliant system where all email is retained for several years, we would expect the amount of duplicate data to be even higher. There is clearly a lot of potential for cost and reliability improvements by managing the unique data, especially given the extended life of reference data, and the need to keep multiple copies of such data to ensure reliable storage. Our analysis further indicates that the effects of duplicate block elimination and compression techniques such as block gzip seem to be relatively independent so that they can be combined to get additive results.

We find that attachments constitute as much as 80% of the email data and that to detect duplicate attachments, it is sufficient to compare attachments with the same name. However, many attachments that are not exact duplicates of another do share many common blocks so that duplicate block detection using the sliding blocking method is a lot more effective at discovering duplicate data than comparing whole attachments. We also discover that a duplicate block is most likely to occur within a few days of its previous occurrence.

8 Acknowledgements

We would like to thank Fred Douglis for making the Users data set available to us.

References

[1] *Enterprise Email Archiving: Market Problems, Needs and Trends*. Osterman Research, Black Diamond, WA, Sept. 2003.

[2] *Reference Information: The Next Wave*. Enterprise Storage Group, Milford, MA, 2003.

[3] BRODER, A. Some applications of rabin's fingerprinting method. In *Sequences II: Methods in Communications, Security, and Computer Science*, R. Capocelli, A. D. Santis, and U. Vaccaro, Eds. Springer-Verlag, 1993, pp. 143–152.

[4] DEUTSCH, P. Gzip file format specification version 4.3. Tech. rep., Aladdin Enterprises, 1996.

[5] DOUGLIS, F., AND IYENGAR, A. Application-specific delta-encoding via resemblance detection. In *Usenix Annual Technical Conference* (June 2003), pp. 113–126.

[6] GANGER, G. R., MCKUSICK, M. K., SOULES, C. A. N., AND PATT, Y. N. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems 18*, 2 (May 2000), 127–153.

[7] MORETON, T. D., PRATT, I. A., AND HARRIS, T. L. Storage, mutability and naming in pasta. In *International Workshop on Peer-to-Peer Computing* (May 2002).

[8] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A low-bandwidth network file system. In *Symposium on Operating Systems Principles* (2001), pp. 174–187.

[9] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, FIPS 180-1. *Secure Hash Standard*. US Department of Commerce, Apr. 1995.

[10] QUINLAN, S., AND DORWARD, S. Venti: A new approach to archival data storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)* (Monterey, CA, Jan. 2002), pp. 89–102.

[11] RABIN, M. O. Fingerprinting by random polynomials. Tech. Rep. TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[12] TRIDGELL, A., AND MACKERRAS, P. The rsync algorithm. Tech. Rep. TR-CS-96-05, Australian National University, 1996.

[13] VARIAN, H., 2003. Dean of the School of Information Management and Systems, University of California, Berkeley.

[14] ZADOK, E., BADULESCU, I., AND SHENDER, A. Extending file systems using stackable templates. In *Usenix Annual Technical Conference* (1999), pp. 57–70.

[15] ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23*, 3 (1977), 337–343.