Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup

Deepavali Bhagwat
University of California
1156 High Street
Santa Cruz, CA 95064
dbhagwat@soe.ucsc.edu

Kave Eshghi
Hewlett-Packard Laboratories
1501 Page Mill Rd
Palo Alto, CA 94304
kave.eshghi@hp.com

Darrell D. E. Long
University of California
1156 High Street
Santa Cruz, CA 95064
darrell@soe.ucsc.edu

Mark Lillibridge
Hewlett-Packard Laboratories
1501 Page Mill Rd
Palo Alto, CA 94304
mark.lillibridge@hp.com

Abstract—**Data deduplication is an essential and critical component of backup systems. Essential, because it reduces storage space requirements, and critical, because the performance of the entire backup operation depends on its throughput. Traditional backup workloads consist of large data streams with high locality, which existing deduplication techniques require to provide reasonable throughput.**

We present Extreme Binning, a scalable deduplication technique for non-traditional backup workloads that are made up of individual files with no locality among consecutive files in a given window of time. Due to lack of locality, existing techniques perform poorly on these workloads. Extreme Binning exploits *file similarity* instead of locality, and makes only one disk access for chunk lookup *per file*, which gives reasonable throughput. Multi-node backup systems built with Extreme Binning scale gracefully with the amount of input data; more backup nodes can be added to boost throughput. Each file is allocated using a stateless routing algorithm to only one node, allowing for maximum parallelization, and each backup node is autonomous with no dependency across nodes, making data management tasks robust with low overhead.

I. Introduction

The amount of digital information created in 2007 was 281 exabytes; by 2011, it is expected to be 10 times larger [1]. 35% of this data originates in enterprises and consists of unstructured content, such as office documents, web pages, digital images, audio and video files, and electronic mail. Enterprises retain such data for corporate governance, regulatory compliance [2], [3], litigation support, and data management.

To mitigate storage costs associated with backing up such huge volumes of data, data deduplication is used. Data deduplication identifies and eliminates duplicate data. Storage space requirements can be reduced by a factor of 10 to 20 or more when backup data is deduplicated [4].

Chunk-based deduplication [5]–[7], a popular deduplication technique, first divides input data streams into fixed or variable-length chunks. Typical chunk sizes are 4 to 8 kB. A cryptographic hash or *chunk ID* of each chunk is used to determine if that chunk has been backed up before. Chunks with the same chunk ID are assumed identical. New chunks are stored and references are updated for duplicate chunks. Chunk-based deduplication is very effective for backup workloads, which tend to be files that evolve slowly, mainly through small changes, additions, and deletions [8].

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

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

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

Existing approaches exploit this 'chunk locality' to improve deduplication throughput. Zhu *et al.* [9] store and prefetch groups of chunk IDs that are likely to be accessed together with high probability. Lillibridge *et al.* [10] batch up chunks into large segments, on the order of 10 MB. The chunk IDs in each incoming segment are sampled and the segment is deduplicated by comparing with the chunk IDs of only a few carefully selected backed up segments. These are segments

[0]Original published in *Proceedings of the 17th IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems* (MASCOTS'2009), London, UK, September 2009. Copyright 2009 by IEEE.

that share many chunk IDs with the incoming segment with high probability.

We now consider the case for backup systems designed to service fine-grained low-locality backup workloads. Such a workload consists of files, instead of large data streams, that arrive in random order from disparate sources. We assume no locality between files that arrive in a given window of time. Several scenarios generate such a workload: File backup and restore requests made by Network-attached Storage (NAS) clients; Continuous Data Protection (CDP), where files are backed up as soon as they have changed; and electronic emails that are backed up as soon as they are received. In fact, NAS-based backup systems that provide a NFS/CIFS interface are already being sold by companies such as NetApp, Data Domain, and EMC. In absence of locality, existing approaches perform poorly: either their throughput [9] or their deduplication efficiency [10] deteriorates.

Our solution, Extreme Binning, exploits file similarity instead of chunk locality. It splits the chunk index into two tiers. One tier is small enough to reside in RAM and the second tier is kept on disk. Extreme Binning makes a single disk access for chunk lookup per file instead of per chunk to alleviate the disk bottleneck problem. In a distributed setting, with multiple *backup nodes* – nodes that perform file based backup – every incoming file is allocated, using a stateless routing algorithm, to a single backup node only. Backup nodes are autonomous – each node manages its own index and data without sharing or knowing the contents of other backup nodes. To our knowledge, no other approach can be scaled or parallelized as elegantly and easily as Extreme Binning.

One disadvantage of Extreme Binning compared to approaches by Zhu *et al.* [9] and Rhea *et al.* [11] is that it allows some duplicate chunks. In practice, however, as shown by our experiments, this loss of deduplication is minimal for representative workloads, and is more than compensated for by the low RAM usage and scalability of our approach.

II. Chunk-based Deduplication

Chunking divides a data stream into fixed [7] or variable length chunks. Variable-length chunking has been used to conserve bandwidth [6], to weed out near duplicates in large repositories [5], for wide-area distributed file systems [12], and, to store and transfer large directory trees efficiently and with high reliability [13].

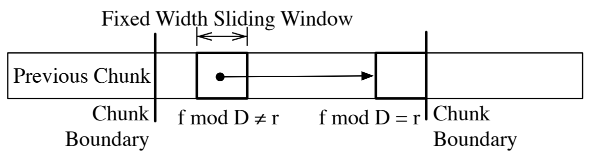


Fig. 1. Sliding Window Technique

Figure 1 depicts the sliding window chunking algorithm. To chunk a file, starting from the beginning, its contents as seen through a fixed-sized (overlapping) sliding window are examined. At every position of the window, a fingerprint or signature of its contents, f, is computed using hashing techniques such as Rabin fingerprints [14]. When the fingerprint, f, meets a certain criteria, such as $f \bmod D = r$ where D, the divisor, and r are predefined values; that position of the window defines the boundary of the chunk. This process is repeated until the complete file has been broken down into chunks. Next, a cryptographic hash or chunk ID of the chunk is computed using techniques such as MD5 [15], or SHA [16], [17].

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

How much deduplication is obtained depends on the inherent content overlaps in the data, the granularity of chunks and the chunking method [18], [19]. In general, smaller chunks yield better deduplication.

III. Our Approach: Extreme Binning

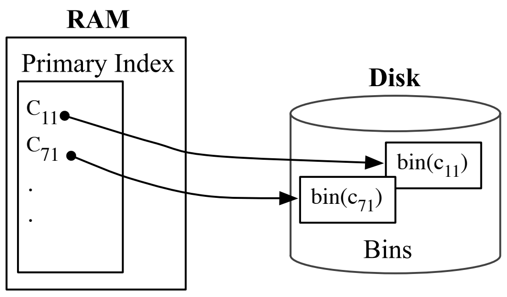


Fig. 2. A two-tier chunk index with the primary index in RAM and bins on disk

Extreme Binning splits up the chunk index into two tiers. The top tier called the primary chunk index or *primary index* resides in RAM. The primary index contains one chunk ID entry *per file*. This chunk ID is the *representative chunk ID* of the file. The rest of the file's chunk IDs are kept on disk in the second tier which is a mini secondary index that we call *bin*. Each representative chunk ID in the primary index contains a pointer to its bin. This two-tier index has been depicted in Figure 2. The two-tier design distinguishes Extreme Binning from some of the other approaches [9], [11] which use a *flat chunk index* – a monolithic structure containing the chunk IDs of all the backed up chunks.

A. Choice of the Representative Chunk ID

The success of Extreme Binning depends on the choice of the representative chunk ID for every file. This choice is governed by Broder's theorem [20]:

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

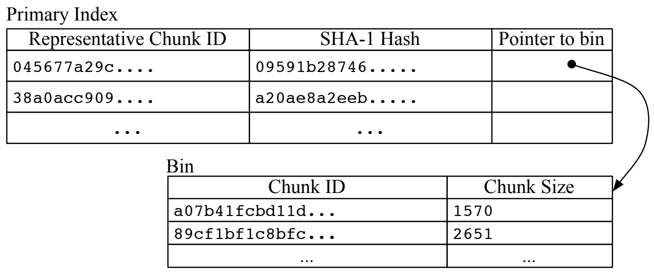
$$\Pr\big[\min(H(S_1)) = \min(H(S_2))\big] = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$$

Broder's theorem states that the probability that the two sets S_1 and S_2 have the same minimum hash element is the same as their Jaccard similarity coefficient [21]. So, if S_1 and S_2 are highly similar then the minimum element of $H(S_1)$ and $H(S_2)$ is the same with high probability. In other words, if two files are highly similar they share many chunks and hence their minimum chunk ID is the same with high probability. Extreme Binning chooses the minimum chunk ID of a file to be its representative chunk ID.

Our previous work [22] has shown that Broder's theorem can be used to identify similar files with high accuracy without using brute force methods. In this previous work, chunk IDs extracted from each file within the given corpus were replicated over multiple partitions of the search index. Each partition was expected to fit in RAM. Such a solution is not feasible in the context of large scale backup systems that can be expected to hold exabytes of data, and are the focus of this work.

Extreme Binning extends our previous work in that it applies those techniques to build a scalable, parallel deduplication technique for such large scale backup systems. Our contributions in this work are: the technique of splitting the chunk index into two tiers between the RAM and the disk to achieve excellent RAM economy, the partitioning of the second tier into bins, and the method of selecting only one bin per file to amortize the cost of disk accesses without deteriorating deduplication. Together, these contributions extend our previous work considerably.

B. File Deduplication and Backup using Extreme Binning


Primary Index

Representative Chunk ID	SHA-1 Hash	Pointer to bin
045677a29c....	09591b28746.....	
38a0acc909....	a20ae8a2eeb.....	
...	...	

Bin

Chunk ID	Chunk Size
a07b41fcbd11d...	1570
89cf1bf1c8bfc...	2651
...	...

Fig. 3. Structure of the primary index and the bins

Figure 3 shows the structure of the primary index and one bin. There are three fields in the primary index – the representative chunk ID, the whole file hash and a pointer to a bin. Each bin contains two fields: the chunk ID and the chunk size. In addition, bins may also contain other metadata, such as the address of the corresponding chunk on disk. File backup proceeds as follows:

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

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

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

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

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

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

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

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

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

IV. A Distributed File Backup System using Extreme Binning

To distribute and parallelize file backup using multiple backup nodes, the two-tier chunk index must first be partitioned and each partition allocated to a backup node. To do this, every entry in the primary index is examined to determine to which backup node it should go. For example, if there are K backup nodes, then every representative chunk ID c_i, in the primary index, is allocated to backup node $c_i \bmod K$. Techniques such as RUSH [23] or LH* [24] can also used for this distribution. These techniques are designed to distribute objects to maximize scalability and reliability.

When a primary index entry moves, the bin attached to it also moves to the same backup node. For every chunk ID in the bin there exists a corresponding data chunk. All the data chunks attached to the bin also move to the same backup node. Each bin is independent. The system has no knowledge of any common chunk IDs in different bins. This means that if a chunk ID appears in two bins, there will be two copies of its corresponding data chunk. Hence, moving a primary index entry to a new backup node, along with its bin and data chunks, does not create any dependencies between the backup nodes. Even if more backup nodes are added in the future to scale out the backup system, the bins and their corresponding chunks can be redistributed without generating any new dependencies. This makes scale out operations clean and simple.

The architecture of a distributed file backup system built using Extreme Binning has been shown in Figure 4. It consists of several backup nodes. Each backup node consists of a compute core and RAM along with a dedicated attached disk. The RAM hosts a partition of the primary index. The corresponding bins and the data chunks are stored on the attached disk as shown in the figure.

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

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

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

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

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

V. Experimental Setup

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

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

To evaluate Extreme Binning on a non-proprietary and impartial data set, we have also tested it on widely available

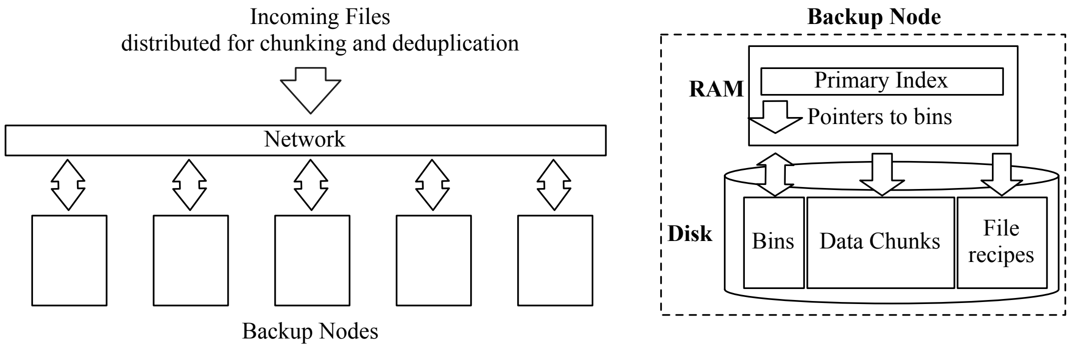


Fig. 4. Architecture of a Distributed File Backup System build using Extreme Binning

Linux distributions. This 26 GB data set consisted of 450 versions from version 1.2.1 to 2.5.75.

It has been shown that data deduplication is more effective when there is low variance in chunk sizes. Consequently, we used the Two Threshold Two Denominators (TTTD) [5] to chunk files. TTTD has been shown to perform better than the basic sliding window chunking algorithm in finding duplicate data and reducing the storage overheads of chunk IDs and their metadata [25]. The average size of the chunks was 4 KB. The chunks were not compressed. Any other chunking algorithm can also be used.

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

Our approach simply enables one to perform scalable efficient searches for duplicate chunks using chunk IDs. Once it has been determined that another chunk with the same chunk ID exists in the backup store, it is always possible to actually fetch the chunk and do a byte by byte comparison instead of a compare by hash [26], [27] (chunk ID) approach, though, at the cost of reduced deduplication throughput as reported by Rhea *et al.* [11].

Our hardware setup was: each backup node ran Red Hat Enterprise Linux AS release 4 (Nahant Update 5). HP MPI (Message-Passing Interface), which contains all the MPI-2 functionality, was used for inter-process communication. Berkeley DB v.4.7.25 [28] was used for the primary index and the bins.

VI. Results

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

A. Deduplication Efficiency

Figure 5 shows how Extreme Binning did while finding duplicates in HDup. The three curves show how much storage space was consumed when there was no deduplication, if every duplicate chunk was identified (perfect deduplication), and when Extreme Binning was used. A small number of duplicate

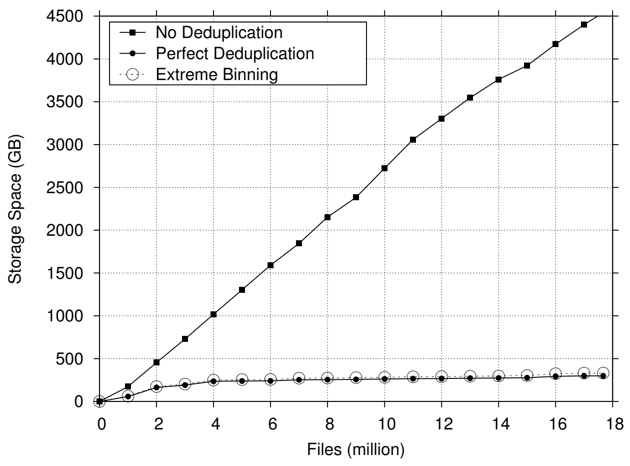


Fig. 5. Deduplication for HDup

chunks are allowed by Extreme Binning. With perfect deduplication, the storage space utilization was 299.35 GB (space reduction ratio: 15.16:1), whereas with Extreme Binning it was 331.69 GB (space reduction ratio: 13.68:1). Though Extreme Binning used an extra 32.3 GB, this overhead is very small compared to the original data size of 4.4 TB.

Figure 6 shows Extreme Binning's deduplication efficiency for LDup. Perfectly deduplicated, LDup data set required 288.03 GB (space reduction ratio: 3.33:1) whereas Extreme Binning consumed 315.09 GB (space reduction ratio: 3.04:1). From these graphs, it is clear that Extreme Binning yields excellent deduplication and that the overhead of extra storage space is small.

Figure 7 shows similar results for the Linux data set. The distributions were ordered according to the version number before being deduplicated. Every point on the x-axis corresponds to one version; there are a total of 450 versions. Perfectly deduplicated data size was 1.44 GB (space reduction ratio: 18.14:1) while with Extreme Binning it was 1.99 GB (space reduction ratio: 13.13:1). In this case too, Extreme Binning yields very good deduplication.

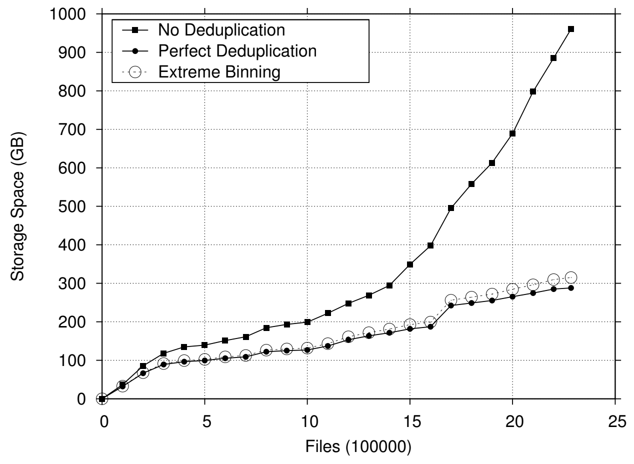


Fig. 6. Deduplication for LDup

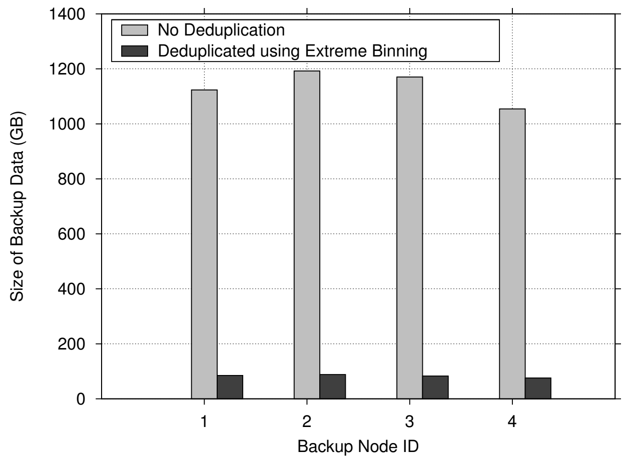


Fig. 8. Size of backup data on each backup node for HDup when using 4 backup nodes

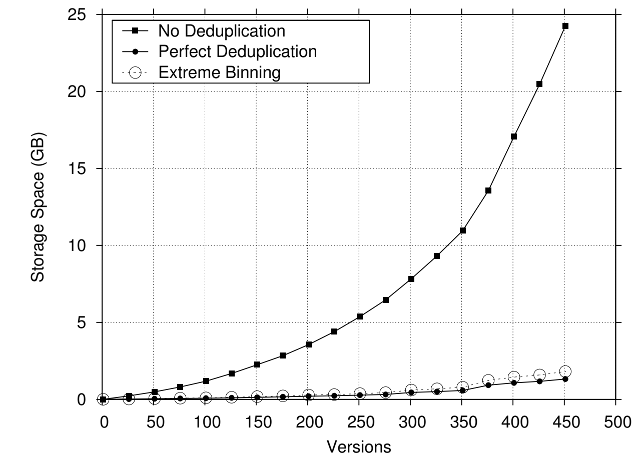


Fig. 7. Deduplication for Linux

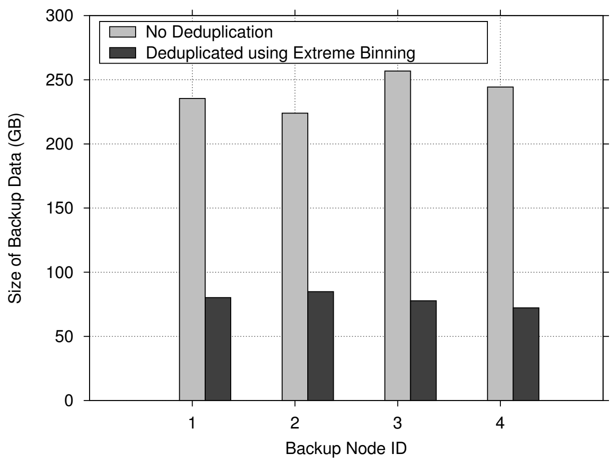


Fig. 9. Size of backup data on each backup node for LDup when using 4 backup nodes

B. Load Distribution

Figures 8 and 9 show how much data, both deduplicated and otherwise, is managed by each backup store for HDup and LDup respectively. It is clear that no single node gets overloaded. In fact, the deduplicated data is distributed fairly evenly. The same trend was observed when 2 through 8 nodes were used. This means that the distribution of files to backup nodes is not uneven. This property of Extreme Binning is vital towards ensuring smooth scale out and preventing any node from becoming a bottleneck to the overall system performance.

Figure 10 depicts the load distribution of Extreme Binning for Linux when 4 nodes were used. Once again, the same trend was observed when 2 through 8 nodes were used.

C. Comparison with Distributed Hash Tables

A flat chunk index could be partitioned like a DHT; by using a consistent hashing scheme to map every chunk ID to a partition. Every partition can then be hosted by a dedicated compute node. To deduplicate a file, the same hashing scheme can be used to dictate which partition should be queried to ascertain if the corresponding chunk is duplicate. Assume a file containing n chunks is to be deduplicated and that the flat chunk index has been partitioned into P partitions such that $n > P$. In the worst case, *all* P partitions will have to queried to deduplicate this file. Our experiments using the DHT technique have shown that for HDup, with $P = 4$, for 52% of the files more than 1 partition was queried and for 27% all 4 partitions were queried. For Linux distributions too, for 50% of the files more than 1 partition was queried and for 12% all 4 partitions were queried. Such a wide query fanout to deduplicate a file reduces the degree of parallelization – fewer files can be deduplicated at the same time. Further, such an infrastructure cannot be used to design a decentralized backup system where every backup node autonomously manages the indices and the data. It is not clear *which* node in the DHT-like scheme stores the deduplicated file given that the file's duplicate chunks may spread across more than one node. Autonomy of backup nodes is not possible in such a design. Further, partitioning the flat chunk index does not reduce RAM

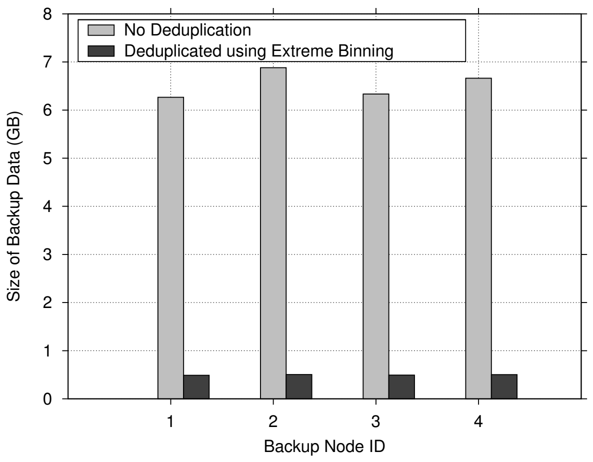


Fig. 10. Size of backup data on each backup node for Linux when using 4 backup nodes

requirements. The total RAM usage remains the same whether the index is partitioned or not. Because Extreme Binning only keeps a subset of all the chunk IDs in RAM, its RAM requirement is much lower than a flat chunk index. This means that fewer backup nodes will be required by Extreme Binning than a DHT like scheme to maintain throughput.

D. RAM Usage

Each primary index entry consisted of the representative chunk ID (20 bytes), the whole file hash (20 bytes), and a pointer to its corresponding bin. The size of the pointer will depend on the implementation. For simplicity we add 20 bytes for this pointer. Then, every record in the primary index is 60 bytes long. With only one backup node, the RAM usage for Extreme Binning was 54.77 MB for HDup (4.4 TB). Even with the overheads of the data structure, such as a hash table used to implement the index, the RAM footprint is small. For the same data set a flat chunk index would require 4.63 GB. The ratio of RAM required by the flat chunk index to that required by Extreme Binning is 86.56:1. For LDup this ratio is 83.70:1, which proves that Extreme Binning provides excellent RAM economy.

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

VII. Related Work

Two primary approaches have been previously proposed for handling deduplication at scale: sparse indexing [10] and that of Bloom filters with caching [9].

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

Zhu *et al.*'s approach [9] always produces perfect deduplication but relies heavily on inherent data locality for its cache to be effective to improve throughput. This approach uses an in-memory Bloom filter [29] and caches index fragments, where each fragment indexes a set of chunks found together in the input. The lack of chunk locality renders the caching ineffectual and each incoming new version of an existing chunk requires reading an index fragment from disk.

Foundation [11] is a personal digital archival system that archives nightly snapshots of user's entire hard disk. By archiving snapshots or disk images, Foundation preserves all the dependencies within user data. Our approach is distinct from Foundation in that Extreme Binning is designed to service fine grained requests for individual files rather than nightly snapshots.

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

DeDe [30] is a decentralized deduplication technique designed for SAN clustered file systems that support a virtualization environment via a shared storage substrate. Each host maintains a write-log that contains the hashes of the blocks it has written. Periodically, each host queries and updates a shared index for the hashes in its own write-log to identify and reclaim duplicate blocks. Deduplication is done out-of-band so as to minimize its impact on file system performance. Extreme Binning, on the other hand, is designed for in-line deduplication and, in a distributed environment the backup nodes do not need to share any index between them. Rather, each backup node deduplicates files independently, using its own primary index and bins only, while still being able to achieve excellent deduplication.

Chunk-based storage systems detect duplicate data at granularities that range from entire file, as in EMC's Centera [31], down to individual fixed-size disk blocks, as in Venti [7] and variable-size data chunks as in LBFS [6]. Variable-width chunking has also been used in the commercial sector, for example, by Data Domain and Riverbed Technology. Deep Store [32] is a large-scale archival storage system that uses three techniques to reduce storage demands: content-addressable storage [31], delta compression [33], [34] and chunking [6].

Delta compression with byte-by-byte comparison, instead of hash based comparison using chunking, has been used for the design of a similarity based deduplication system [35]. Here, the incoming data stream is divided into large, 16 MB blocks, and sampled. The samples are used to identify other, possibly similar blocks, and a byte-by-byte comparison is conducted to remove duplicates.

Distributed Hashing is a technique for implementing efficient and scalable indices. Litwin *et al.* [24] proposed Scalable Distributed Data Structures based on linear hash tables for parallel and distributed computing. Distributed Hash Tables (DHT) have also been widely used in the area of peer-to-peer systems [36] and large-scale distributed storage systems [37], [38] to distribute and locate content without having to perform exhaustive searches across every node in the system. Chord [39], Pastry [40] and Tapestry [41] are some of the DHT implementations used in a distributed environment.

VIII. Future Work

To achieve high throughput, backup systems need to write new chunks sequentially to disk. Chunks related to each other – because they belong to the same file or the same directory – are not stored together. Due to this, restoration and retrieval requires a non-trivial number of random disk seeks. In the worst case, reads require one disk seek per chunk, along with any additional seeks required to find the location of the chunk by accessing the appropriate index. When the retrieval request is for a large set of data, for example, a user's home directory, these disk seeks will slow down the retrieve operation considerably. This is unacceptable, especially when lost data needs to be restored quickly, as in the case of disaster recovery, and time is of essence. However, if chunks that are expected to be retrieved together can be grouped together while being written to disk, then fewer random seeks will be required. Our future work will consist of methods that reduce data fragmentation and disk latencies during restores.

IX. Conclusions

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

Acknowledgments

The authors would like to thank Jehan-François Pâris and Ethan Miller for their helpful comments. Darrell Long and Deepavali Bhagwat were supported in part by the Kumar Malavalli Endowment, by the Petascale Data Storage Institute under Department of Energy Office of Science award DE-FC02-06ER25768, Hewlett-Packard Laboratories, and by the Storage Systems Research Center member companies.

References

[1] J. F. Gantz, C. Chute, A. Manfrediz, S. Minton, D. Reinsel, W. Schlichting, and A. Toncheva, "The diverse and exploding digital universe: An updated forecast of worldwide information growth through 2011," IDC, An IDC White Paper - sponsored by EMC, March 2008.

[2] "104th Congress, United States of America. Public Law 104-191: Health Insurance Portability and Accountability Act (HIPAA)," August 1996.

[3] "107th Congress, United States of America. Public Law 107-204: Sarbanes-Oxley Act of 2002," July 2002.

[4] H. Biggar, "Experiencing Data De-Duplication: Improving Efficiency and Reducing Capacity Requirements," *The Enterprise Strategy Group*, Feb. 2007.

[5] G. Forman, K. Eshghi, and S. Chiocchetti, "Finding similar files in large document repositories," in *KDD '05: Proceeding of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, 2005, pp. 394–400.

[6] 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)*, 2001, pp. 174–187.

[7] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage," in *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST)*, 2002, pp. 89–101.

[8] 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*, Apr. 2004. [Online]. Available: you-mss04.pdf

[9] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST)*, 2008, pp. 269–282.

[10] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Campbell, "Sparse Indexing: Large scale, inline deduplication using sampling and locality," in *Proceedings of the Eighth USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2009, pp. 111–123.

[11] S. Rhea, R. Cox, and A. Pesterev, "Fast, inexpensive content-addressed storage in Foundation," in *Proceedings of the 2008 USENIX Annual Technical Conference*, 2008, pp. 143–156.

[12] S. Annapureddy, M. J. Freedman, and D. Mazières, "Shark: Scaling file servers via cooperative caching," in *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, 2005, pp. 129–142.

[13] K. Eshghi, M. Lillibridge, L. Wilcock, G. Belrose, and R. Hawkes, "Jumbo Store: Providing efficient incremental upload and versioning for a utility rendering service," in *Proceedings of the Fifth USENIX Conference on File and Storage Technologies (FAST)*, 2007, pp. 123–138.

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

[15] R. Rivest, "The MD5 message-digest algorithm," IETF, Request For Comments (RFC) 1321, Apr. 1992. [Online]. Available: http://www.ietf.org/rfc/rfc1321.txt

[16] National Institute of Standards and Technology, "Secure hash standard," FIPS 180-1, Apr. 1995. [Online]. Available: http://www.itl.nist.gov/fipspubs/fip180-1.htm

[17] National Institute of Standards and Technology, "Secure hash standard," FIPS 180-2, Aug. 2002. [Online]. Available: http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

[18] C. Policroniades and I. Pratt, "Alternatives for detecting redundancy in storage systems data," in *Proceedings of the General Track: 2004 USENIX Annual Technical Conference*, 2004, pp. 73–86.

[19] M. Dutch, "Understanding data deduplication ratios," *SNIA Data Management Forum*, June 2008.

[20] A. Z. Broder, "On the resemblance and containment of documents," in *SEQUENCES '97: Proceedings of the Compression and Complexity of Sequences 1997*, 1997, pp. 21–29.

[21] P. Jaccard, "Étude comparative de la distribution orale dans une portion des Alpes et des Jura," *In Bulletin del la Société Vaudoise des Sciences Naturelles*, vol. 37, pp. 547–579, 1901.

[22] D. Bhagwat, K. Eshghi, and P. Mehra, "Content-based document routing and index partitioning for scalable similarity-based searches in a large corpus," in *KDD '07: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2007, pp. 105–112.

[23] R. J. Honicky and E. L. Miller, "Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution," in *Proceedings of the 18th International Parallel & Distributed Processing Symposium (IPDPS 2004)*, vol. 1, 2004, pp. 96–.

[24] W. Litwin, M.-A. Neimat, and D. A. Schneider, "LH* — a scalable, distributed data structure," *ACM Transactions on Database Systems*, vol. 21, no. 4, pp. 480–525, 1996.

[25] K. Eshghi, "A framework for analyzing and improving content-based chunking algorithms," Hewlett Packard Laboraties, Palo Alto, Tech. Rep. HPL-2005-30(R.1), 2005.

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

[27] J. Black, "Compare-by-hash: a reasoned analysis," in *Proceedings of the Systems and Experience Track: 2006 USENIX Annual Technical Conference*, 2006, pp. 85–90.

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

[29] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[30] A. T. Clements, I. Ahmad, M. Vilayannur, and J. Li, "Decentralized deduplication in SAN cluster file systems," in *Proceedings of the 2009 USENIX Annual Technical Conference*, Jan. 2009.

[31] EMC Corporation, "EMC Centera: Content Addressed Storage System, Data Sheet," Apr. 2002.

[32] 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)*, Apr. 2005, pp. 804–815.

[33] M. Ajtai, R. Burns, R. Fagin, D. D. E. Long, and L. Stockmeyer, "Compactly encoding unstructured inputs with differential compression," *Journal of the Association for Computing Machinery*, vol. 49, no. 3, pp. 318–367, May 2002.

[34] F. Douglis and A. Iyengar, "Application-specific delta-encoding via resemblance detection," in *Proceedings of the 2003 USENIX Annual Technical Conference*, Jun. 2003, pp. 113–126.

[35] L. Aronovich, R. A. Ron, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein, "The design of a similarity based deduplication system," in *SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. New York, NY, USA: ACM, 2009.

[36] 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)*, 2002, pp. 285–298.

[37] 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 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Nov. 2000, pp. 190–201.

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

[39] 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)*, 2001, pp. 149–160.

[40] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, November 2001, pp. 329–350.

[41] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 41–53, January 2004.