Tradeoffs in Scalable Data Routing for Deduplication Clusters

Wei Dong*
Princeton University

Fred Douglis
EMC

Kai Li
Princeton University and EMC

Hugo Patterson
EMC

Sazzala Reddy
EMC

Philip Shilane
EMC

Abstract

As data have been growing rapidly in data centers, deduplication storage systems continuously face challenges in providing the corresponding throughputs and capacities necessary to move backup data within backup and recovery window times. One approach is to build a cluster deduplication storage system with multiple deduplication storage system nodes. The goal is to achieve scalable throughput and capacity using extremely high-throughput (e.g. 1.5 GB/s) nodes, with a minimal loss of compression ratio. The key technical issue is to route data intelligently at an appropriate granularity.

We present a cluster-based deduplication system that can deduplicate with high throughput, support deduplication ratios comparable to that of a single system, and maintain a low variation in the storage utilization of individual nodes. In experiments with dozens of nodes, we examine tradeoffs between *stateless* data routing approaches with low overhead and *stateful* approaches that have higher overhead but avoid imbalances that can adversely affect deduplication effectiveness for some datasets in large clusters. The stateless approach has been deployed in a two-node commercial system that achieves 3 GB/s for multi-stream deduplication throughput and currently scales to 5.6 PB of storage (assuming 20X total compression).

1 Introduction

For business reasons and regulatory requirements [14, 29], data centers are required to backup and recover their exponentially increasing amounts of data [15] to and from backup storage within relatively small windows of time; typically a small number of hours. Furthermore, many copies of the data must be retained for potentially long periods, from weeks to years. Typically, backup software aggregates files into multi-gigabyte "tar" type files for storage. To minimize the cost of storing the many backup copies of data, these files have traditionally been stored on tape.

Deduplication is a technique for effectively reducing the storage requirement of backup data, making disk-based backup feasible. Deduplication replaces identical regions of data (files or pieces of files) with references (such as a SHA-1 hash) to data already stored on disk [6, 20, 27, 36]. Several commercial storage systems exist that use some form of deduplication in combination with compression (such as Lempel-Ziv [37]) to store hundreds of terabytes up to petabytes of original (logical) data [8, 9, 16, 25]. One state-of-the-art single-node deduplication system achieves 1.5 GB/s in-line deduplication throughput while storing petabytes of backup data with a combined data reduction ratio in the range of 10X to 30X [10].

To meet increasing requirements, our goal is a backup storage system large enough to handle *multiple* primary storage systems. An attractive approach is to build a deduplication cluster storage system with individual high-throughput nodes. Such a system should achieve scalable throughput, scalable capacity, and a cluster-wide data reduction ratio close to that of a single very large deduplication system. Clustering storage systems [5, 21, 30] are a well-known technique to increase capacity, but adding deduplication nodes to such clusters suffer from two problems. First, it will fail to achieve high deduplication because such systems do not route based on data content. Second, tightly-coupled cluster file systems often do not exhibit linear performance scalability because of requirements for metadata synchronization or fine-granularity data sharing.

Specialized deduplication clusters lend themselves to a loosely-coupled architecture because consistent use of content-aware data routing can leverage the sophisticated single-node caching mechanisms and data layouts [36] to achieve scalable throughput and capacity while maximizing data reduction. However, there is a tension between deduplication effectiveness and

*Work done in part as an intern with Data Domain, now part of EMC.

throughput. On one hand, as chunk size decreases, deduplication rate increases, and single-node systems may deduplicate chunks as small as 4-8 KB[1] to achieve very high deduplication. On the other hand, with larger chunk sizes, high throughput is achieved because of stream and inter-file locality, and per-chunk memory overhead is minimized [18, 35]. High throughput deduplication with small chunk sizes is achieved on individual nodes using techniques that take advantage of cache locality to reduce I/O bottlenecks [20, 36]. For existing deduplication clusters like HYDRAstor [8], though, relatively large chunk sizes (∼64 KB) are used to maintain high throughput and fault tolerance at the cost of deduplication. We would like to achieve scalable throughput and capacity with cluster-wide deduplication close to that of a state-of-the-art single node.

In this paper, we propose a deduplicating cluster that addresses these issues by intelligently "striping" large files across a cluster: we create *super-chunks* that represent consecutive smaller chunks of data, *route* super-chunks to nodes, and then perform deduplication at each node. We define data routing as the assignment of super-chunks to nodes. By routing data at the granularity of super-chunks rather than individual chunks, we maintain cache locality, reduce system overheads by batch processing, and exploit the deduplication characteristics of smaller chunks at each node. The challenges with routing at the super-chunk level are, first, the risk of creating duplicates, since the fingerprint index is maintained independently on each node; and second, the need for scalable performance, since the system can overload a single node by routing too much data to it.

We present two techniques to solve the data routing problem in building an efficient deduplication cluster, and we evaluate them through trace-driven simulation of collected backups up to 50 TB. First, we describe a *stateless* technique that routes based on only 64 bytes from the super-chunk. It is remarkably effective on typical backup datasets, usually with only a ∼10% decrease in deduplication for small clusters compared to a single node; for balanced workloads the gap is within ∼10-20% even for clusters of 32–64 nodes. Second, we compare the stateless approach to a *stateful* technique that uses information about where previous chunks were routed. This achieves deduplication nearly as high as a single node and distributes data evenly among dozens of nodes, but it requires significant computation and either greater memory or communication overheads. We also explore a range of techniques for routing super-chunks that trade off memory and communication requirements, including varying how super-chunks are formed, how large they are on average, how they are assigned to nodes, and how node imbalance is addressed.

The rest of this paper is organized as follows. Section 2 describes our system architecture, then Section 3 focuses on alternatives for super-chunk creation and routing. Section 4 presents our experimental methodology, datasets, and simulator, and Section 5 shows the corresponding results. We briefly describe our product in Section 6. We discuss related work in Section 7, and conclusions and future work are presented in Section 8.

2 System Overview

This section presents our deduplication cluster design. We first review the architecture of our earlier storage system [36], which we use as a single-node building block. Because the design of the single-node system emphasizes **high throughput**, any cluster architecture must be designed to support scalable performance. We then show the design of the deduplication cluster with stateless routing, corresponding to our product (differences pertaining to *stateful* routing are presented later in the paper).

We use the following criteria to govern our design decisions for the system architecture and choosing a routing strategy:

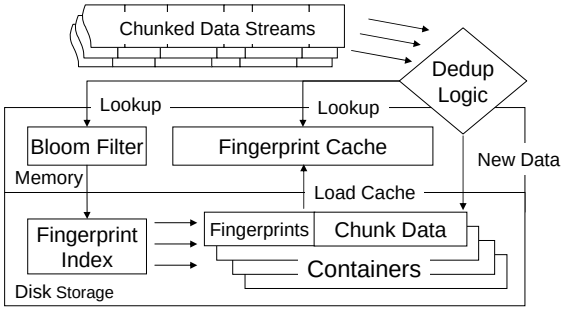
- **Throughput** Our cluster should scale throughput with the number of nodes by maximizing parallel usage of high-throughput storage nodes. This implies that our architecture must optimize for cache locality, even with some penalty with respect to deduplication capacity—we will write duplicates across nodes for improved performance, within reason.
- **Capacity** To maximize capacity, repeated patterns of data should be forwarded to storage nodes in a consistent fashion. Importantly, capacity usage should be balanced across nodes, because if a node fills up, the system must place new data on alternate nodes. Repeating the same data on multiple nodes leads to poor deduplication.

The architecture of our single-node deduplication system is shown in Figure 1(a). We assume the incoming data streams have been divided into chunks with a content-based chunking algorithm [4, 22], and a fingerprint has been computed to uniquely identify each chunk. The main task of the system is to quickly determine whether each incoming chunk is new to the system and then to efficiently store new chunks. High-throughput fingerprint lookup is achieved by exploiting the *deduplication locality* of backup datasets: in the same backup stream, chunks following a duplicate chunk are likely to be duplicates, too.

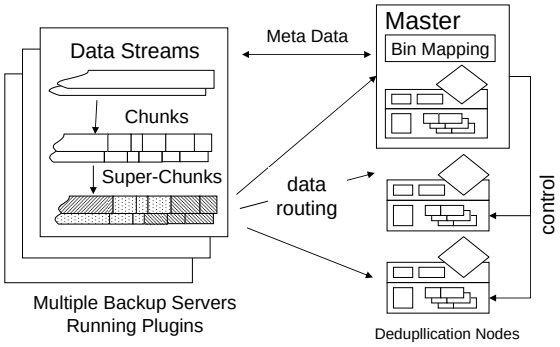
To preserve locality, we use a technique based on Stream Informed Segment[2] Layout [36]: disk storage is

[1] Throughout the paper, references to chunks of a given size refer to chunks that are expected to *average* that size.

[2] Note that the term "segment" in the earlier paper means the same



(a) Deduplication Node Architecture



(b) Dataflow of Deduplication Cluster

Figure 1: Deduplication node architecture and cluster design using individual nodes as building blocks.

divided into fixed-size large pieces called containers, and each stream has a dedicated container. The non-duplicate fingerprints and chunk data are appended to the metadata part and the data part of the container. The sequence of fingerprints needed to reconstruct a file is also written as chunks and stored to containers, and a root fingerprint is maintained in a directory structure. When the current container is full, it is flushed to disk, and a new container is allocated for the stream.

To identify existing chunks, a fingerprint cache avoids a substantial fraction of index lookups, and for those not found in the cache, a Bloom filter [3] identifies with high probability which fingerprints will be found in the on-disk index. Thus disk accesses only occur either when a duplicate chunk misses in our cache or when a full container of new chunks is flushed to disk. (In rare cases, a false positive from the Bloom filter will cause an unnecessary lookup to the on-disk index.) Once a fingerprint is loaded, many fingerprints that were written at the same time are loaded with it, enabling subsequent duplicate chunks to hit in the fingerprint cache.

Figure 1(b) demonstrates how to combine multiple deduplication nodes into a cluster. Backup software on each client collects individual files into a backup stream, which it transfers to a backup server. We offer a plugin [12] that runs on a customer's backup servers, which divides each stream into chunks, fingerprints them, groups them into a super-chunk, and routes each super-chunk to a deduplicating storage node. Each storage node locally applies deduplication logic to chunks while preserving data locality, which is essential to maintain high throughput.

To clarify the parallelization that takes place in our cluster, consider writing a file to the cluster. When a super-chunk is routed to a storage node, deduplication begins while the next super-chunk is created and routed to a potentially different node. All of the metadata needed to reconstruct a file is stored in chunks and distributed across the nodes. When reading back a file, parallel reads are initiated to all of the nodes by looking ahead through the metadata references and issuing reads for super-chunks to the appropriate nodes. To achieve maximum parallelization, the I/O load should be equal on each node, and both read and write throughput should scale linearly with the number of nodes.

Note that we do not yet specifically address the inter-node dependencies that arise in the event of a failure. Each node is highly redundant, with RAID and other data integrity mechanisms. It would be possible to provide redundant controllers in each node to eliminate that single point of failure, but these details are beyond the scope of this paper.

Storage Rebalancing: When super-chunks are routed to a storage node, we use a level of indirection called a *bin*. We assign a super-chunk to a bin using the `mod` function, and then map each bin to a given node. By using many more bins ($\sim$1000) than actual nodes, the Bin Manager (running on the master node) can rebalance nodes by reassigning bins in the future. The Bin Manager also handles expansion cases such as when a node's storage expands or when a new node is added to the cluster. In those cases, the Bin Manager reassigns bins to the new storage to maintain balanced usage. Rebalancing data takes place online while backups and other operations continue, and the entire process is transparent to the user. After a rebalance operation, the cluster will generally remain balanced for future backups. The master node communicates the bin-to-node mapping to the plugin.

Bin migration occurs when the storage usage of a node exceeds the average usage in the cluster by some threshold (defaulting to 5%). Note that if there is a great deal of skew in the total physical storage of a single bin, that bin can exceed the threshold even if it is the only bin stored on a node. Such anomalous behavior is rare but possible, and we discuss some examples of this in Section 5.

as the term "chunk" in this paper.

3 Data Routing

This section addresses two issues with data routing in our deduplication cluster: how to group chunks into super-chunks, and how to route data. Super-chunk formation is relatively straightforward and is discussed in Section 3.1. We focus here on two routing strategies: stateless routing, light-weight and well suited for most balanced workloads (Section 3.2); and stateful routing, requiring more overhead but maintaining a higher deduplication rate with larger clusters (Section 3.3).

3.1 Super-Chunk Formation

There are two important criteria for grouping consecutive chunks into super-chunks. First, we want an average super-chunk size that supports high throughput. Second, we want super-chunk selection to be resistant to small changes between full backups.

The size of a super-chunk could vary from a single chunk to many megabytes, or it could be equal to individual files as suggested by Extreme Binning [2]. We experimented with a variety of average super-chunk sizes from 8 KB up to 4 MB on backup datasets. The average super-chunk size affects deduplication, balance across storage nodes, and throughput, and it is more thoroughly explored in Section 5.3. We generally found that a 1 MB average super-chunk size is a good choice, because it results in efficient data locality on storage nodes as well as generally high deduplication, and this is the default value used in our experiments unless otherwise noted.

Determining super-chunk boundaries (anchoring) mirrors the problem of anchoring chunks [24] in many ways and should be implemented in a content-dependent fashion. Since all chunks in a super-chunk are routed together, deduplication is affected by super-chunk boundaries. We represent each chunk with a feature (see the next subsection), compare the feature against a mask, and when the mask is matched, the selected chunk becomes the boundary between super-chunks. Minimum and maximum super-chunk sizes are enforced, half and double the desired super-chunk size respectively.

3.2 Stateless Routing

Numerous data routing techniques are possible: routing based only on the contents of the current super-chunk is *stateless*, while routing super-chunks using information about the location of existing chunks is *stateful* (see Section 3.3).

For stateless routing, the basic technique is to produce a feature value representing the data and then apply a simple function (such as `mod` #bins) to the value to make the assignment. As a super-chunk is a sequence of chunks, we first compute a feature from each chunk, and then select one of those features to represent the super-chunk.

There are many options for generating a chunk feature. A hash could be calculated over an entire chunk (`hash(*)`) or over a prefix of the bytes near an anchor point (`hash(N)`, for a prefix of N bytes). Using the hash of a representative portion of a chunk results in data that are similar, but not identical, being routed to the same node; the net effect is to improve deduplication while increasing skew. We tried a range of prefix lengths and found the best results when using the first 64 bytes after a chunk anchor point (*i.e.*, `hash(64)`), which we compare to `hash(*)`. When using a hash for routing rather than deduplication, collisions are acceptable, so we use the first 32-bit word of the SHA-1 hash for `hash(64)`.

In addition, we considered other variants, such as fingerprints computed over sliding windows of content [22]; these did not make a substantial difference in the outcome, and we do not discuss them further.

To select a super-chunk feature based on the chunk features, the first, maximum, minimum, or most common chunk feature could be selected; using just the first has the advantage that it is not necessary to buffer an entire super-chunk before deciding where to route it, something that matters when hundreds or thousands of streams are being processed simultaneously. Another stateless technique is to treat the feature of each chunk as a "vote" for a node and select the most common, which does not work especially well, because hash values are often uniformly distributed. We experimented with a variety of options and found the most interesting results with four combinations: `hash(64)` of the first chunk, the minimum `hash(64)` across a super-chunk, `hash(*)` of the first chunk, and the minimum `hash(*)` across a super-chunk (compared in detail in Section 5.2). Elsewhere, `hash(64)` refers to the feature from the first chunk unless stated otherwise.

The main advantages of stateless techniques are (1) reduced overhead for recording node assignments, and (2) reduced requirements for recovering this state after a system failure. Stateless routing has some properties of a "shared nothing" [31] architecture because of limited shared state. There is a potential for a loss of deduplication compared to the single-node case, and there is also the potential for increased data skew if the selected features are not uniformly distributed. We find empirically that the reduction in deduplication effectiveness is within acceptable bounds, and bin migration can usually address excessive data skew.

3.3 Stateful routing

Using information about the location of existing chunks can improve deduplication, at an increased cost in (a) computation and (b) memory or communication. We present a stateful approach that produces deduplication that is frequently comparable to that of a single node

even with a significant number of nodes (32-64); also, by balancing the benefit of matching existing chunks against the capacity of overloaded nodes, it avoids the need to migrate data after the fact. This approach is not a panacea, however, as it increases memory requirements (per-node Bloom filters, if storing them on a master node, and buffering an entire super-chunk before routing it) and computational overhead, as discussed below.

To summarize our stateful routing algorithm, in its simplest form:

1. Use a Bloom filter to count the number of times each fingerprint in a super-chunk is already stored on a given node.

2. Weight the number of matches ("votes") by each node's relative storage utilization. Overweight nodes are excluded.

3. If the highest weighted vote is above a threshold, select that node.

4. If no node has sufficient weighted votes, route to the node selected via `hash(64)` of the first chunk if it is not overloaded; otherwise route to the least loaded node.

We now explain the algorithm in more detail. To route a super-chunk, once the master node knows the number of chunks in common with (a.k.a. "matching") each node, it selects a destination. However, such a "voting" approach requires care to avoid problematic cases: simply targeting the node with the most matching chunks will route more and more super-chunks there, because the more data it has relative to other nodes, the more likely it is to match the most chunks.

Thus, one refinement to this stateful approach is to create a threshold for a minimum fraction of chunks that must match a node before it is selected. With a uniform distribution, one expects each node to match at most $\frac{C}{N}$ chunks on average, where C is the number of chunks in the super-chunk and N is the number of nodes. Typically not all chunks will match any node, and the average number of matches will be lower, but if a node already stores significantly more than the expected average, this is a reason to route the super-chunk to that node. In our system, a *voting benefit threshold* of 1.5 means that a node is considered as a candidate only if it already matches at least $\frac{1.5C}{N}$ chunks. This prevents a node from being selected simply because it matches more than any other node, when no node matches well enough to be of interest.

Simply using a static threshold for the number of matches to vote a super-chunk to a particular node still results in high data skew, as popular nodes get more popular over time. A technique we call **weighted voting** addresses that deficiency by striking a balance between deduplication and uniform storage utilization. It decreases the perceived value of known duplicates in proportion to the extent to which a node is overloaded relative to the average storage utilization of the system. As an example, if a node matches $\frac{2C}{N}$ chunks in a super-chunk, but that node stores 120% ($\frac{6}{5}$) of the average node, then the node is treated as though it matched $\frac{5}{6} * \frac{2C}{N}$ chunks. Note that while a node that stores less than the average could be given a weight < 1, increasing the overall weighted value, instead we assign such nodes a weight of 1. This ensures that when multiple nodes can easily accommodate the new super-chunk, the node is selected based on the best match. We experimented with various weight functions, but we found that it is effective simply to exclude nodes that are above a capacity threshold. In practice, a capacity of 5% above the average was selected as the threshold (see Sec 5.4).

The computational cost arises because the stateful approach computes where every chunk in a super-chunk is currently stored. A Bloom filter lookup has to be performed for each chunk, on each node in the cluster, before a routing destination can be picked. Each such lookup is extremely fast ($\sim 100 - 200$ns), but there can be a great many of these lookups: inserting M chunks into an N-node cluster would result in NM Bloom filter lookups, compared to M lookups in a single-node system. The additional overhead in memory or communication depends on whether the master node(s) tracks the state of each storage node (resulting in substantial memory allocations) or sends the chunk fingerprints to the storage nodes and collects counts of how many chunks match each node (resulting in communication overhead). One way to mitigate the effect is to *sample* [20] chunks that are used for voting. We reduce the number of chunks considered by checking each chunk's fingerprint for a bit pattern of a specific length (e.g., B bits must match a pattern for a $1/2^B$ sampling rate); the total number of lookups is then approximately $NM/2^B$. Without sampling, the total cost of the Bloom filter lookups is about 1.2 hours of computation for a 5-TB dataset, but a sampling rate of $1/8$ cuts this to 13 minutes of overhead with a nominal reduction in deduplication (see Section 5.5). That work can further be parallelized across back-ends or in threads on the front-end.

As an example, the general approach to weighted voting is depicted in Figure 2. In this example, the seven numbered chunks in this super-chunk are sampled for voting. Chunks 1, 3, and 4 are contained on node 1, chunks 2, 3, 5, and 6 are on node 2, chunk 5 is also on node 4, and chunk 7 is not stored on any node. Node 1 has 3 raw votes, and node 2 has 4. Factoring in space, since node 2 uses much more than the average,

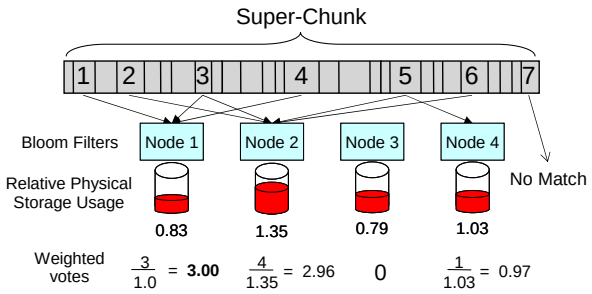


Figure 2: Weighted voting example. A node with many matches will be selected if it does not also have too much data already, relative to the other nodes. Any node with a relative storage usage of less than 1 is treated as though it is at the average.

its weighted votes are $(4/1.35) = 2.96$. Node 1 has a slightly higher weighted vote of 3. The minimum weight for a node to be selected is $\frac{1.5\times7}{4} = 2.6$. Thus node 1 is selected for routing.

The main advantage of a stateful technique is the opportunity to incorporate expected deduplication and capacity balancing while assigning chunks to nodes. On the other hand, computational or communication overhead must be considered when choosing this technique, though it is an attractive option for coping with unbalanced workloads or cluster sizes beyond our current expectations.

4 Experimental Methodology

We use trace-driven simulation to evaluate the tradeoffs of the various techniques described in the previous section. This section describes the datasets used, the evaluation metrics, and the details of the simulator.

4.1 Datasets

In this paper, we simulate super-chunk routing for nine datasets. Three were collected from large backup environments representing typical scenarios where a backup server hosts multiple data types from dozens of clients. These datasets contain approximately 40-50 TB precompressed data. To analyze how our routing technique handles datasets with specific properties, we also analyze five datasets representing single data types. Four of the datasets are each approximately 5 TB and a fifth is about 13 TB. In addition, we synthesize a "blended" dataset consisting of a mixture of the five smaller datasets. In general, we use them in the form that a deduplication appliance would see them: tar files that are usually many gigabytes in size, rather than individual small files. With the exception of the "blended" dataset, all of these datasets represent real backups from production environments.

Name	Size (GB)		Dedup.	Months
	Total	Peak		
Collection 1	40,695	2,867	6.1	1–2
Collection 2	44,536	1,536	11.5	4–6
Collection 3	51,584	2,150	6.1	3
Perforce	4,574	250	20.8	6
Workstations	4,926	200	5.6	6
Exchange	5,253	33	6.8	7
System Logs	5,436	122	38.7	4
Home Dirs.	12,907	855	19.3	3
Blended	33,097	N/A	12.5	N/A

Table 1: Summary of datasets. The Collection datasets were collected from backup servers with multiple data types, and the other datasets were collected from single data-type environments. Deduplication ratios are obtained from a single-node system.

For the three collected datasets, we received permission to analyze production backup servers within EMC. We gathered traces for each file including the timestamp, sequence of chunk fingerprints, and other metadata necessary to analyze chunk routing. At an earlier collection on internal backup servers, we gathered copies of backup files for the individual data types.

Table 1 lists salient information of these datasets: the total logical size, the daily peak size, the single-node deduplication rate, and the number of months in the 99th percentile of retention period. The datasets are:

Collection 1: Backups from approximately 100 clients consisting of half software development and half business records. Backups are retained 1-2 months.

Collection 2: Backups from approximately 50 engineering workstations with 4 months of retention and servers with 6 months of retention.

Collection 3: Backups of over 100 clients for Exchange, SQL servers, and Windows workstations with 3 months of retention.

Perforce: Backups from a version control repository.

Workstations: Backups from 16 workstations used for build and test.

Exchange: Backups from a Microsoft Exchange server. Each day contains a single full backup.

System Logs: Backups from a server's `/var` directory, containing numerous system files and logs. Full backups were created weekly.

Home Directory: Backups from engineers' home directories, containing source code, office documents, etc. Full backups were created weekly.

Blended: To explore the effects of multiple datasets being written to a storage system (a common scenario), we created a blended dataset. We combined alternating super-chunks of the single data-type datasets, weighted

by overall size; thus there are approximately two super-chunks from the "home directory" dataset for each super-chunk of the others. The overall deduplication for this dataset (12.5) is somewhat higher than the weighted average across the datasets (12.3), due to some cross-dataset commonality.

While our experiments studied all of these datasets, because of space limitations, we typically only present results for two: `Workstations` and `Exchange`. Experiments with `Workstations` have results consistent with the other datasets and represents our expected customer experience. The `Exchange` dataset showed consistently worse results with our techniques and is presented for comparison. Because of data patterns within `Exchange`, using a 1-MB super-chunk results in overloading a single bin with $1/16$ of the data.

4.2 Evaluation Metrics

The principal evaluation metrics are:

Total Deduplication (TD): The ratio of the original dataset size to the size after identical chunks are eliminated. (We do not consider *local compression* (e.g., Lempel-Ziv [37]), which is orthogonal to the issues considered in this paper.)

Data Skew: The ratio of the largest node's physical (post-deduplication) storage usage to the average usage, used to evaluate how far from this perfect balance a particular configuration is. High skew leads to a node filling up and duplicate data being written to alternative nodes, as discussed in Section 2.

Effective Deduplication (ED): Total Deduplication divided by Data Skew, as a single utility measure that encompasses both deduplication effectiveness and storage imbalance. ED is equivalent to Total Deduplication computed as if every node consumes the same amount of physical storage as the most loaded node. This metric is meaningful because the whole cluster degrades when one node is filled up. ED permits us to compare routing techniques and parameter options with a single value.

Normalized ED: ED divided by deduplication achieved by a single-node system. This is an indication of how close a super-chunk routing method is to the ideal deduplication achievable on a cluster system. It allows us to compare the effectiveness of chunk-routing methods across different datasets under the same $[0, 1]$ scale.

Fingerprint Index Lookups: Number of on-disk index lookups, used as an approximation to throughput. The lookup *rate* is the number of lookups divided by the number of chunks processed by a storage node.

4.3 Simulator

Most of the results presented in this paper come from a set of simulations, organized as follows:

1. For the Collection datasets, we read from a deduplicating storage node and reconstructed files based on metadata to create a full trace including the chunk size, its `hash(*)` value, and its `hash(64)` value. The other datasets were preprocessed by reading in each file, computing chunks of a particular average size (typically 8 KB), and storing a trace.

2. The per-chunk data are passed into a program to determine super-chunk boundaries and route those super-chunks to particular nodes. It produces statistical information about deduplication rates, data skew, the number of Bloom filter lookups performed, and so on. In addition, it logs the SHA1 hash and location of each super-chunk, on a per-node basis. Its parameters include the super-chunk routing algorithm; the average super-chunk size (typically **1 MB**); the maximum relative node size before bin migration is performed (for stateless) or node assignment is avoided (for stateful), defaulting to **1.05**; some stateful routing parameters described below, and several others not considered here.

The simulator was validated in part by comparing deduplication results for Total Deduplication and skew to the values reported by the live two-node system. Due to minor implementation differences, normalized TD is typically up to 2–3% higher in the simulator than in the live system, though in one case the real system reported slightly higher normalized deduplication. Skew is similarly close.

The stateful routing parameters are: (a) Vote sampling: what fraction of chunks, on average, should be passed to the Bloom filters and checked for matches? (Default: **1/8**.) (b) Vote threshold: how many more matches than the average (as a fraction) should an average-sized node be, before being used rather than the node routed by the first chunk? (Default: **1.5**)

3. To analyze caching effects on a storage system, each of the node-specific super-chunk files can be used to synthesize a data stream with the same deduplication patterns and chunk sizes, which speeds up experimentation relative to reading the original data repeatedly. For simplicity, the compression for the synthesized chunks was fixed at 2:1, a close approximation to overall compression for the datasets used. This stream is then written to a deduplication appliance, sending each bin to its final node in the original simulations after migration.

The accuracy of using a synthesized stream in place of the original dataset was validated by comparing Total Deduplication of several synthesized results to those of original datasets.

5 Experimental Results

We focused our experiments on analyzing the impact of super-chunk routing on capacity and fingerprint index lookups across a range of cluster sizes and a variety of datasets. We start by surveying how different routing ap-

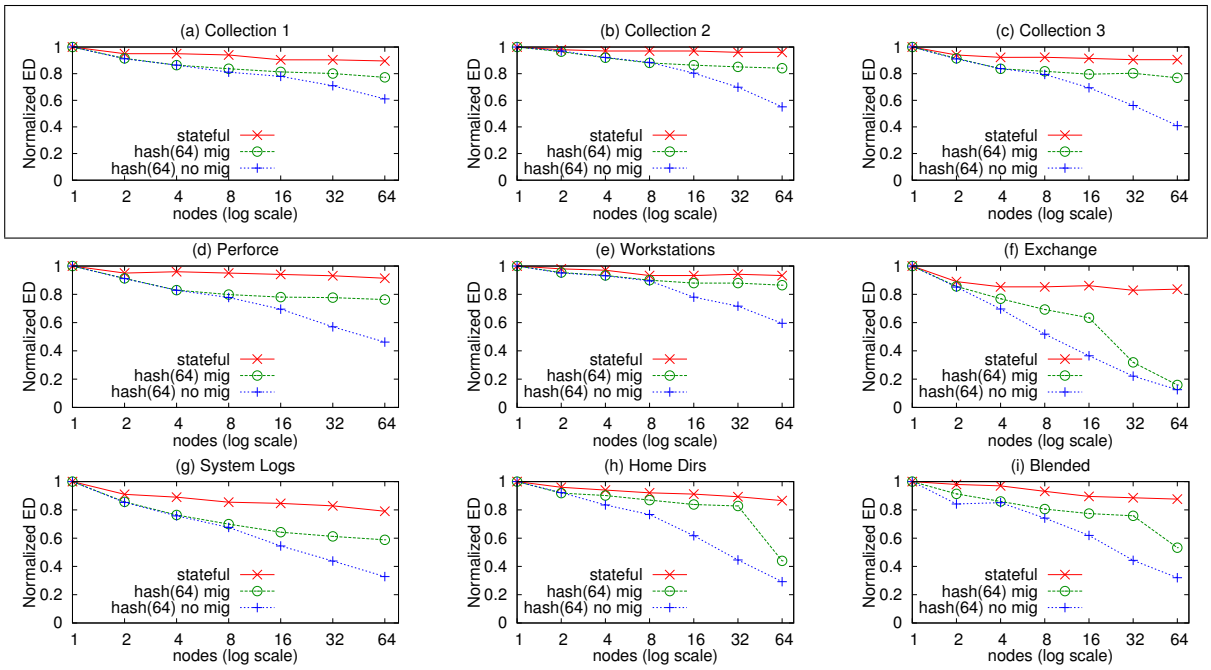


Figure 3: Normalized ED of the stateless and stateful techniques as a function of the number of nodes. The top row represents the collected "real-world" datasets. Stateful and `hash(64)` (mig) use a capacity threshold of 5%.

proaches fare over a broad range of datasets and cluster sizes (Section 5.1). This gives a picture of how Total Deduplication and skew combine into the Effective Deduplication metric. Then we dive into specifics:

- What is the best feature (`hash(64)` vs. `hash(*)`, routing by first chunk vs. all chunks in a super-chunk) for routing super-chunks (Section 5.2)?
- How does super-chunk size affect fingerprint cache lookups and locality (Section 5.3)?
- How sensitive is the system to various parameter settings, including capacity threshold (Section 5.4) and those involved in stateful routing (Section 5.5)?

5.1 Overall Effectiveness

We first compare the basic techniques, stateless and stateful, across a range of datasets. Figure 3 shows a scatter plot for the nine datasets and three algorithms: `hash(64)` without bin migration, `hash(64)` with a 5% migration threshold, and stateful routing with a 5% capacity limitation.

In general, `hash(64)` without migration works well for small clusters (2–4 nodes) but degrades steadily as the cluster size increases. Adding bin migration greatly improves the ED for most of the datasets, though even with bin migration, ED for `Exchange` decreases rapidly as the number of nodes increases, and there is also a sharp decrease for `Home Directories` and `Blended` at 64 nodes. This skew occurs when a single bin is substantially larger than the average node utilization (see Section 5.4). Stateful routing is often within 10% of the single-node deduplication even at 64 nodes, although for some datasets the gap is closer to 20%. However, there is additional overhead, as discussed in Section 5.5.

Table 2 presents normalized Total Deduplication (TD), data skew, and Effective Deduplication (ED) for several datasets, as the number of nodes varies (corresponding to the `hash(64)` (mig) and stateful curves in Figure 3). It shows how a moderate increase in skew results in a moderate reduction in ED (`Workstations`), but `Exchange` suffers from both repeated data (losing $\frac{1}{3}$ of TD) and significant skew (further reducing ED by a factor of 4).

5.2 Feature Selection

As discussed in Section 3.2, there are a number of ways to route a super-chunk. Here we compare four super-chunk features: `hash(64)` of the *first* chunk, the minimum of all `hash(64)`, the `hash(*)` of the first chunk, or the minimum of all `hash(*)`. We also compare against the method used by HYDRAstor [8], which consists of 64-KB chunks routed based on their fingerprint. Figure 4 shows the normalized ED of these four features for two datasets, not factoring in any capacity limitations. For `Workstations`, all four choices are similarly effective, which is consistent with the other datasets that are not

# nodes	hash(64) TD	hash(64) Skew	hash(64) ED	stateful TD	stateful Skew	stateful ED
Collection 1						
1	1.00	1.00	1.00	1.00	1.00	1.00
2	0.93	1.02	0.91	0.95	1.00	0.95
4	0.89	1.03	0.86	0.95	1.00	0.95
8	0.86	1.03	0.84	0.95	1.01	0.94
16	0.85	1.04	0.81	0.94	1.04	0.91
32	0.83	1.04	0.80	0.94	1.04	0.91
64	0.83	1.07	0.77	0.94	1.05	0.90
Collection 2						
1	1.00	1.00	1.00	1.00	1.00	1.00
2	0.97	1.00	0.97	0.98	1.00	0.98
4	0.94	1.02	0.92	0.97	1.00	0.97
8	0.92	1.04	0.88	0.97	1.00	0.97
16	0.90	1.04	0.86	0.97	1.00	0.97
32	0.88	1.04	0.85	0.96	1.00	0.96
64	0.87	1.04	0.84	0.96	1.00	0.96
Collection 3						
1	1.00	1.00	1.00	1.00	1.00	1.00
2	0.92	1.01	0.92	0.95	1.01	0.94
4	0.88	1.05	0.84	0.95	1.03	0.93
8	0.85	1.04	0.82	0.96	1.04	0.92
16	0.84	1.05	0.80	0.96	1.05	0.91
32	0.83	1.03	0.80	0.95	1.05	0.91
64	0.82	1.07	0.77	0.95	1.05	0.91
Workstations						
1	1.00	1.00	1.00	1.00	1.00	1.00
2	0.97	1.02	0.95	0.98	1.00	0.98
4	0.95	1.02	0.93	0.98	1.01	0.97
8	0.94	1.04	0.90	0.98	1.04	0.94
16	0.92	1.05	0.88	0.98	1.04	0.94
32	0.91	1.04	0.88	0.97	1.03	0.94
64	0.91	1.05	0.86	0.97	1.04	0.93
Exchange						
1	1.00	1.00	1.00	1.00	1.00	1.00
2	0.86	1.01	0.86	0.89	1.00	0.89
4	0.78	1.01	0.77	0.87	1.02	0.85
8	0.72	1.04	0.69	0.87	1.02	0.85
16	0.68	1.08	0.63	0.87	1.01	0.86
32	0.67	2.09	0.32	0.87	1.05	0.83
64	0.65	4.12	0.16	0.87	1.04	0.83

Table 2: Total Deduplication (TD), data skew, and normalized Effective Deduplication ratio ($ED = \frac{TD}{skew}$) for some of the datasets, using capacity thresholds of 5%.

shown. `Exchange` demonstrates the extreme case, in which most chunk-routing features degrade badly with large clusters. One can see the effect of high skew when a common feature results in distinct chunks being routed to the same node. This is less common when the entire chunk's hash is used than when a prefix is used: `first hash(*)` spreads out the data more, resulting in less data skew and better ED. Even though chunks are consistently routed with the HYDRAstor technique (`HYDRAstor`), the ED is generally worse than the other techniques because of the larger chunk size: the deduplication is less than half that achieved with 8-KB chunks on a single node.

The figure demonstrates that `first hash(64)` is generally somewhat better for smaller clusters, while `first hash(*)` is better for larger ones. (This effect arises because `first hash(64)` is more likely to keep putting even somewhat similar chunks on the same node, which improves deduplication but increases skew.) Using the minimum of either feature, as Extreme Binning does for `hash(*)`, generally achieves similar deduplication to using the first chunk. Due to its effectiveness with the cluster sizes being deployed in the near future and its reduction in buffer requirements, we use `first hash(64)` as the default and refer to it as `hash(64)` for simplicity elsewhere.

5.3 Factors Impacting Cluster Throughput

A major goal of our architecture is to maximize throughput as the cluster scales, and in a deduplicating system, the main throughput bottleneck is fingerprint index lookups that require a random disk read [36]. We are not able to produce a throughput measure in MB/s through simulation, so we use fingerprint index lookups as an indirect measure of throughput.

There are two important issues involving fingerprint index lookups to consider. The first is the total number of fingerprint index lookups that take place, since this is a measure of the amount of work required to process a dataset and is impacted by data skew. The second is the rate of fingerprint index lookup, which indicates the locality of data written to disk. These values are impacted both by the super-chunk size and number of nodes in a cluster, and we have selected a relatively large cluster size (32 nodes) while varying the super-chunk size.

Early generations of backups (the first few weeks of a dataset) tend to be laid out sequentially because of a low deduplication rate, while higher generations of backups are more scattered. To highlight this impact, we analyzed the caching effects while writing the final 1 TB of each synthesized dataset across the N nodes. In these experiments, the cache size is held at 12,500 fingerprints. While this may seem small, it is similar to a cache of 400,000 fingerprints on a single, large node, Also, a cache must handle multiple backup streams, while our experiments use one dataset at a time.

Figure 5 shows the skew of the uncompressed (logical) data, maximum normalized total number of fingerprint index lookups, maximum normalized fingerprint index lookup rate, and ED when routing super-chunks of various sizes for (a) `Workstations` and (b) `Exchange`. Note that we report skew of the logical data here instead of skew of the post-dedupe data reported elsewhere, because fingerprint lookups happen on logical data. The

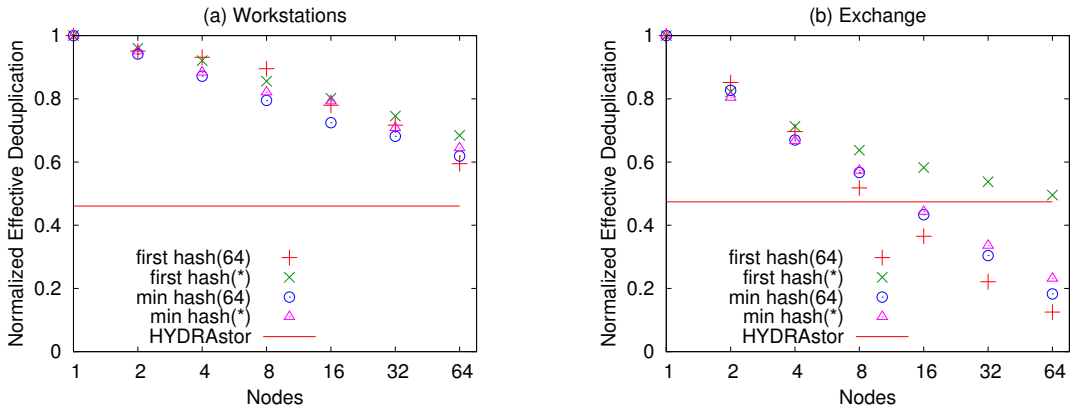


Figure 4: Normalized ED versus number of nodes with various features. No bin migration is performed. The HYDRAstor points represent 64-KB chunks routed without super-chunks, with virtually no data skew but significantly worse deduplication in most cases. *`Workstations` is representative of many other datasets, while `Exchange` is anomalous.*

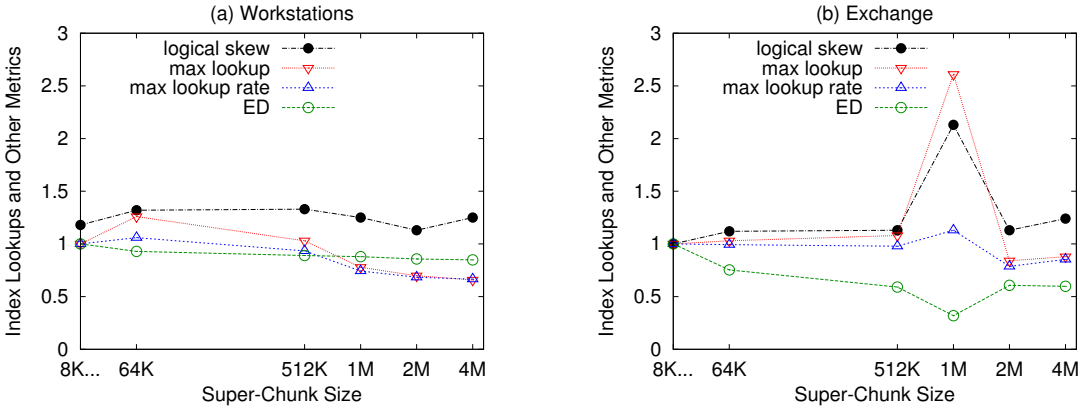


Figure 5: Skew of data written to nodes (pre-deduplication), maximum number of fingerprint index lookups and lookup rate, and ED versus super-chunk size for a 32-node cluster. Fingerprint index lookup values are normalized relative to those metrics when routing individual 8-KB chunks. As the super-chunk size increases, the maximum number of on-disk index lookups decreases for `Workstations` (improving throughput), while effective deduplication decreases. *`Workstations` is representative of many other datasets, while `Exchange` is anomalous.*

fingerprint index lookup numbers are normalized relative to the rate seen when routing individual 8-KB chunks. Because the lookup rate improvement achieved by using larger super-chunk sizes generally comes with a cost of lower deduplication, we also plot normalized ED to aid the selection of an appropriate super-chunk size. It should be noted that we found smaller differences in lookup rate and total number of lookups with smaller clusters.

For `Workstations`, we see that the total number of fingerprint index lookups and rate generally shrink as we use larger super-chunk sizes. Routing 4-MB super-chunks results in $\sim 65\%$ of the maximum total index lookups compared to routing chunks. Though data skew, maximum lookup rate, and maximum number of lookups tend to follow the same trends, the values for maximum number of lookups and maximum lookup rate may come from different nodes.

The index lookups (both total and rate) for `Exchange` around 1 MB highlights a case where our technique may perform poorly due to a frequently repeating pattern in the data set that causes a large fraction of the `hash(64)` values to map to the same bin. With smaller super-chunk sizes, less data are carried with each super-chunk, so skew can be reasonably balanced via migration, and for larger super-chunks, the problematic `hash(64)` value is no longer selected. For this dataset, a super-chunk size of 1 MB results in higher skew that lowers ED, and it has a high total number of lookups and worst-case cache miss rate. This is a particularly difficult example for our sys-

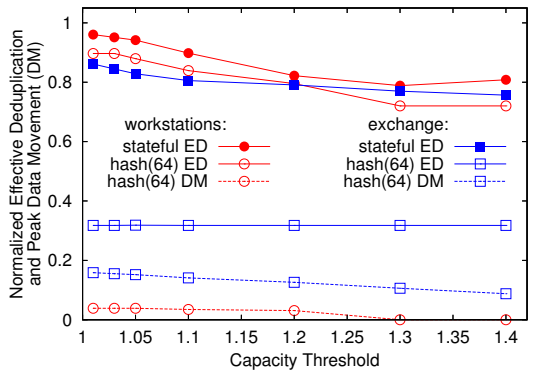


Figure 6: Normalized ED as a function of capacity threshold on 32 nodes, for `hash(64)` and stateful, and peak fraction of data movement (DM) for `hash(64)`. Note that lower points are better for data movement, while higher is better for ED.

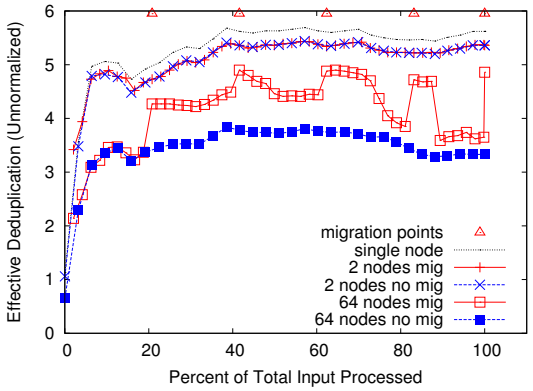


Figure 7: Effective Deduplication as a function of the amount of data processed, with and without bin migration at a 5% threshold, for the `Workstations` dataset on 2 and 64 nodes. Migration points are marked along the top, every 1 TB, with deduplication computed every 0.1 TB. The deduplication for a single node is depicted as the top curve.

tem as the same node had both the highest lookup rate and skew, which roughly multiply together to equal total lookups.

Although any particular super-chunk size can potentially result in skew if patterns in the data result in one bin being selected too often, the problem is rare in practice. Thus, despite this one poor example, we decided that 1-MB super-chunks provide both reasonable throughput and deduplication and use that as the default super-chunk size in our other experiments.

The scalability of our cluster design could more thoroughly be analyzed with a comparison of the number of fingerprint index lookups for various cluster sizes relative to the single node case. Intuitively, a single-node system might have similar lookup characteristics to nodes in a cluster when routing very large super-chunks and without data skew.

5.4 Space Usage Thresholds

Limitations on storage use arise in two contexts. For stateless routing, we periodically migrate bins away from nodes storing more than the average, if they exceed a fixed threshold relative to the mean. In the simulations, bin migration takes place after multiple 1-TB *epochs* have been processed, totaling $\sim 20\%$ of a given dataset. This means that we attempt migrations approximately 5 times per dataset regardless of size, plus once more at the end, if needed. For stateful routing, we refrain from placing new data on a node that is already storing more than that threshold above the average.

Figure 6 demonstrates the impact of the capacity threshold on ED and peak data movement, using the `Workstations` and `Exchange` datasets on 32-nodes. The top four curves show ED: for `Workstations`, deduplication effectiveness improves with increasingly tight capacity bounds, although the benefit below 5% is minimal, while for `Exchange`, the existence of a single oversized bin when using 1 MB super-chunks ensures a large skew regardless of threshold in the case of `hash(64)`.

The bottom two curves provide an indication of the impact of bin migration on data movement, as the threshold changes. We compute the fraction of data moved from a node at the end of an epoch, relative to the amount of physical data stored on the node at the time of the migration, and report the maximum across all nodes and epochs. `Exchange` moves 15–20% of the incoming data (which is on the order of $\frac{1}{32}$ of 1 TB) without improving ED, while we would migrate at most a few percent of one node's data for `Workstations`. Note that across the entire dataset, migration accounts for at most $\frac{1}{1000}$ of the data, and on the 2-node commercial systems currently deployed, they have never occurred. Because at 32 nodes we do see small amounts of migration even for the `Workstations` dataset, and increasing the threshold from 1.01 to 1.05 reduces the total data migrated by nearly a factor of 2 without much of an impact on ED, we use 1.05 as the default threshold in other experiments.

Figure 7 shows the impact of bin migration over time on the `Workstations` dataset. The curves for 2 nodes are identical, as no migration was performed. The curves for 64 nodes are significantly different, with the curve without migration having much worse ED. However, even with migration, the ED drops between migration points due to increasing skew. Note that this graph does not normalize deduplication relative to a single node, in order to highlight the effect of starting with entirely new

Sampling Rate (1-in-N)	Workstations		Exchange		Memory (GB)
	ED	Look-ups (B)	ED	Look-ups (B)	
1	5.28	20.57	5.74	21.95	96
2	5.27	10.83	5.72	11.62	48
4	5.27	6.03	5.76	6.46	24
8	5.31	3.61	5.63	3.88	12
16	5.27	2.41	5.46	2.59	6
32	5.19	1.81	5.13	1.95	3

Table 3: The ED, Bloom filter lookups in billions, and Bloom filter memory requirements in a 32-node system, for two of the datasets. They vary as a function of the sampling rate: which chunks are checked for existence on each node. The memory requirement is independent of the dataset.

data, then increasing deduplication over time.

5.5 Parameters for Stateful Routing

In addition to capacity limitations, stateful routing is parameterized by vote sampling and vote threshold as explained in Section 4.3. Sampling has a great impact on the number of fingerprint lookups, while surprisingly, the system is not very sensitive to a threshold requiring a node to be a particularly good match to be selected.

We evaluated sampling across a variety of datasets and cluster sizes, varying the selectivity of the anchors used to vote from 1 down to $\frac{1}{32}$. Table 3 reports the effect of sampling on ED and Bloom filter lookups for two of the datasets. (Slight rises in ED with less frequent sampling result from slightly *lower* skew due to not matching a node quite as often.) The last column of the table shows the size of a Bloom filter *on a master node* for a 1% false positive rate and up to 20 TB of unique 8-KB chunks on each node; it demonstrates how the aggregate memory requirement on the master would decrease as the sampling rate decreases. The required size to track each node is multiplied by the number of nodes, 32 in this experiment. Each node would also have its own local Bloom filter, which would be unaffected by the sampling rate used for stateful routing. If lookups are forwarded to each node, sampling would be used to limit the number of lookups, but the per-node Bloom filters used for deduplication would be used for routing, and no extra memory would be required.

We found that the ED is fairly constant from looking up all chunks (a sampling rate of 1) down to a rate of $\frac{1}{8}$ and often similar when sampling $\frac{1}{16}$; it degrades significantly, as expected, when less than that. Thus we use a default of $\frac{1}{8}$ for stateful routing elsewhere in this paper.

We also examined the vote benefit threshold. While we use a default of 1.5, the system is not very sensitive to values from 0.75 to 2. The key is to have a high enough threshold that a single chunk will not "attract" more and more dissimilar chunks due to one match.

6 Cluster Deduplication Product

EMC now makes a product based on this technology [13]. The cluster configuration currently consists of two nodes and uses the `hash(64)` routing technique with bin migration. Each node has the following hardware configuration: 4 socket processor, 4 cores per socket, and each core is running at 2.93 Ghz; 64 GB of memory; four 10-Gb Ethernet interfaces (one for external traffic and one for inter-node traffic, both in a fail-over pair); and 140 TB of storage, consisting of 12 shelves of 1-TB drives. Each shelf has 16 drives in a 12+2 RAID-6 configuration with 2 spare drives.

The total physical capacity of the two-node system is 280 TB. Under typical backup usage, total compression is expected to be 20X, which leads to a logical capacity of 5.6 PB of storage. Write performance with multiple streams is over 3 GB/s. Note that this performance was achieved with processing on the backup server as described in Section 2, which communicates with storage nodes to filter duplicate chunks before network transfer. Because of the filtering step, logical throughput (file size divided by transfer time) can even exceed LAN speed.

We measured the steady-state write and read performance with 1–4 nodes and found close to linear improvement as the number of nodes increases. While simulations suggest our architecture will scale to a larger number of nodes, we have not yet tuned our product for a larger system or run performance tests.

In over six months of customer usage, bin migration has never run, which indicates stateless routing typically maintains balance across two nodes.

7 Related Work

Chunk-based deduplication is the most widely used deduplication method for secondary storage. Such a system breaks a data file or stream into contiguous chunks and eliminates duplicate copies by recording references to previous, identical chunks. Numerous studies have investigated content-addressable storage using whole files [1], fixed-size blocks [27, 28], content-defined chunks [17, 24, 36], and combinations or comparisons of these approaches [19, 23, 26, 32]; generally, these have found that using content-defined chunks improves deduplication rates when small file modifications are stored. Once the data are divided into chunks, it is represented by a secure fingerprint (*e.g.*, SHA-1) used for deduplication.

A technique to decrease the in-memory index requirements is presented in Sparse Indexing [20], which uses a sampling technique to reduce the size of the fingerprint

index. The backup set is broken into relatively large regions in a content-defined manner similar to our super-chunks, each containing thousands of chunks. Regions are then deduplicated against a few of the most similar regions that have been previously stored using a sparse, in-memory index with only a small loss of deduplication.

While Sparse Indexing is used in a single system to reduce its memory footprint, the notion of sampling within a region of chunks to identify other chunks against which new data may be deduplicated is similar to our sampling approach in stateful routing. However, we use those matches to direct to a specific node, while they use matches to load a cache for deduplication.

Several other deduplication clusters have been presented in the literature. Bhagwat *et al.* [2] describe a distributed deduplication system based on "Extreme Binning": data are forwarded and stored on a file basis, and the representative chunk ID (the minimum of all chunk fingerprints of a file) is used to determine the destination. An incoming file is only deduplicated against a file with a matching representative chunk ID rather than against all data in the system. Note that Extreme Binning is intended for operations on individual files, not aggregates of all files being backed up together. In the latter case, this approach limits deduplication when inter-file locality is poor, suffers from increased cache misses and data skew, and requires multiple passes over the data when these aggregates are too big to fit in memory.

DEBAR [34] also deduplicates individual files written to their cluster. Unlike our system, DEBAR deduplicates files partially as they are written to disk and completes deduplication during post-processing by sharing fingerprints between nodes.

HYDRAstor [8] is a cluster deduplication storage system that creates chunks from a backup stream and routes chunks to storage nodes, and HydraFS [33] is a file system built on top of the underlying HYDRAstor architecture. Throughput of hundreds of MB/s is achieved on 4-12 storage nodes while using 64 KB-sized chunks. Individual chunks are routed by evenly partitioning fingerprint space across storage nodes, which is similar to the routing techniques used by Avamar [11] and PureDisk [7]. In comparison, our system uses larger super-chunks for routing to maximize cache locality and throughput but also uses smaller chunks for deduplication to achieve higher deduplication.

Choosing the right chunking granularity presents a tradeoff between deduplication and system capacity and throughput even in a single-node system [35]. Bimodal chunking [18] is based on the observation that using large chunks reduces metadata overhead and improves throughput, but large chunks fail to recover some deduplication opportunities when they straddle the point where new data are added to the stream. Bimodal chunking tries to identify such points and uses a smaller chunk size around them for better deduplication.

8 Conclusion and Future Work

This paper presents super-chunk routing as an important technique for building deduplication clusters to achieve scalable throughput and capacity while maximizing effective deduplication. We have investigated properties of both stateless and stateful versions of super-chunk routing. We also describe a two-node deduplication storage product that implements the stateless method to achieve 3 GB/sec deduplication throughput with the capacity to store approximately 5.6 PB of backup data.

Our study has three conclusions. First, we have found that using super-chunks, a multiple of fine-grained deduplication chunks, for data routing is superior to using individual chunks to achieve scalable throughput while maximizing deduplication. We have demonstrated that a 1-MB super-chunk size is a good tradeoff between index lookups, which directly impact deduplication throughput, and effective cluster-wide deduplication.

Second, the stateless routing method (`hash(64)`) with bin migration is a simple and yet efficient way to build a deduplication cluster. Our simulation results on real-world datasets show that this method can achieve good balance and scalable throughput (good caching locality) while achieving at least 80% of the single-node effective deduplication, and bin migration appears to be critical to the success of the stateless approach in larger clusters.

Third, our study shows that effective deduplication of the stateless routed cluster for certain datasets (most notably `Exchange`) may drop quickly as the number of nodes increases beyond 4. To solve this problem, we have proposed a stateful data routing approach. Simulations show this approach can achieve 80% or better normalized ED when using up to 64 nodes in a cluster, even for "pathological" cases.

Several issues remain open. First, we would like to further our understanding of the conditions that cause severe data skew with the stateless approach. To date, no bin migration has occurred in the production system described in this paper; this is not surprising considering that ED for `hash(64)` on two nodes is virtually identical for each of our datasets, with or without bin migration. The same is true for most, but not all, of the datasets as the cluster size increases moderately. Second, we plan to examine the scalability of the system across a broad range of cluster sizes and the impact of parameters such as feature selection and super-chunk size. Third, we want to explore the use of bin migration to support reconfiguration such as node additions. Finally, we plan to build a prototype cluster with stateful routing so that more thorough experiments can be conducted in lab and in customer environments.

Acknowledgments

We thank Dhanabal Ekambaram, Paul Jardetzky, Ed Lee, Dheer Moghe, Naveen Rastogi, Pratap Singh, and Grant Wallace for helpful comments and/or assistance with our experimentation. We are especially grateful to the anonymous referees and our shepherd, Cristian Ungureanu, for their feedback and guidance.

References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: federated, available, and reliable storage for an incompletely trusted environment. In *OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 1–14, 2002.

[2] D. Bhagwat, K. Eshghi, D. D. Long, and M. Lillibridge. Extreme binning: scalable, parallel deduplication for chunk-based file backup. In *MASCOTS 09: Proceedings of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Sept. 2009.

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

[4] S. Brin, J. Davis, and H. García-Molina. Copy detection mechanisms for digital documents. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 398–409, 1995.

[5] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur. Pvfs: a parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 391–430. MIT Press, 2000.

[6] L. P. Cox., C. D. Murray, and B. D. Noble. Pastiche: making backup cheap and easy. In *OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 285–298, New York, NY, USA, 2002. ACM.

[7] M. Dewaikar. Symantec NetBackup PureDisk: optimizing backups with deduplication for remote offices, data center and virtual machines. `http://eval.symantec.com/mktginfo/enterprise/white_papers/b-symantec_ne%tbackup_puredisk_WP-en-us.pdf`, September 2009.

[8] C. Dubnicki, G. Leszek, H. Lukasz, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. HYDRAstor: a scalable secondary storage. In *FAST '09: Proceedings of the 7th conference on File and Storage Technologies*, pages 197–210, February 2009.

[9] EMC Corporation. Data Domain products. `http://www.datadomain.com/products/`, 2009.

[10] EMC Corporation. DD880: deduplication storage for the core data center. `http://www.datadomain.com/pdf/DataDomain-DD880-Datasheet.pdf`, 2009.

[11] EMC Corporation. Efficient data protection with EMC Avamar global deduplication software. `http://www.emc.com/collateral/software/white-papers/h2681-efdta-prot-av%amar.pdf`, July 2009.

[12] EMC Corporation. Data Domain Boost Software, 2010. `http://www.datadomain.com/products/dd-boost.html`.

[13] EMC Corporation. Data Domain Global Deduplication Array, 2010. `http://www.datadomain.com/products/global-deduplication-array.html`.

[14] European Parliament. Directive 2006/24/EC "On the retention of data generated or processed in connection with the provision of publicly available electronic communications services or of public communication networks" , March 2006.

[15] 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. An IDC White Paper — sponsored by EMC, March 2008.

[16] IBM Corporation. IBM ProtecTIER Deduplication Solutions, 2010. `http://www-03.ibm.com/systems/storage/tape/protectier`.

[17] N. Jain, M. Dahlin, and R. Tewari. Taper: tiered approach for eliminating redundancy in replica synchronization. In *FAST '05: Proceedings of the 4th USENIX Conference on File and Storage Technologies*, pages 21–21, 2005.

[18] E. Kruus, C. Ungureanu, and C. Dubnicki. Bimodal content defined chunking for backup streams. In *FAST '10: Proceedings of the 8th Conference on File and Storage Technologies*, February 2010.

[19] P. Kulkarni, F. Douglis, J. LaVoie, and J. M. Tracey. Redundancy elimination within large collections of

files. In *Proceedings of the USENIX Annual Technical Conference*, pages 59–72, 2004.

[20] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *FAST '09: Proceedings of the 7th Conference on File and Storage Technologies*, pages 111–123, 2009.

[21] The Lustre File System, 2010. `http://www.lustre.org`.

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

[23] D. T. Meyer and W. J. Bolosky. A Study of Practical Deduplication. In *FAST '11: Proceedings of the 9th Conference on File and Storage Technologies*, February 2011.

[24] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *SOSP '01: Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 174–187, 2001.

[25] Network Appliance. NetApp ONTAP. `http://www.netapp.com/us/products/platform-os/dedupe.html`, 2009.

[26] C. Policroniades and I. Pratt. Alternatives for detecting redundancy in storage systems data. In *Proceedings of the USENIX Annual Technical Conference*, pages 73–86, 2004.

[27] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *FAST '02: Proceedings of the 1st USENIX conference on File and Storage Technologies*, 2002.

[28] S. C. Rhea, R. Cox, and A. Pesterev. Fast, inexpensive content-addressed storage in foundation. In *Proceedings of the USENIX Annual Technical Conference*, pages 143–156, 2008.

[29] 107th Congress, United States of America. Public Law 107-204: "Sarbanes-Oxley Act of 2002", July 2002.

[30] S. R. Soltis, T. M. Ruwart, and M. T. Okeefe. The global file system. In *MSS '96: Proceedings of the 5th NASA Goddard Conference on Mass Storage*, pages 319–342, 1996.

[31] M. Stonebraker. The case for shared-nothing. *IEEE Database Engineering*, 9(1), March 1986.

[32] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, T. Bressoud, and A. Perrig. Opportunistic Use of Content Addressable Storage for Distributed File Systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 127–140, 2003.

[33] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra. HydraFS: a high throughput file system for the HYDRAstor content-addressable storage system. In *FAST '10: Proceedings of the 8th Conference on File and Storage Technologies*, February 2010.

[34] T. Yang, D. Feng, Z. Niu, K. Zhou, and Y. Wan. DEBAR: a scalable high-performance deduplication storage system for backup and archiving. In *IEEE International Symposium on Parallel & Distributed Processing*, May 2010.

[35] L. You and C. Karamanolis. Evaluation of efficient archival storage techniques. In *MSS '04: Proceedings of the 21st Symposium on Mass Storage Systems*, Apr. 2004.

[36] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *FAST '08: Proceedings of the 6th Conference on File and Storage Technologies*, pages 269–282, February 2008.

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