Data Domain Cloud Tier: Backup here, backup there, deduplicated everywhere!

Abhinav Duggal, Fani Jenkins, Philip Shilane, Ramprasad Chinthekindi, Ritesh Shah, and Mahesh Kamat, *Dell EMC*

https://www.usenix.org/conference/atc19/presentation/duggal

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

Data Domain Cloud Tier: Backup here, backup there, deduplicated everywhere!

Abhinav Duggal Fani Jenkins Philip Shilane Ramprasad Chinthekindi Ritesh Shah
Mahesh Kamat
Dell EMC

Abstract

Data Domain has added a cloud tier capability to its on-premises storage appliance, allowing clients to achieve the cost benefits of deduplication in the cloud. While there were many architectural changes necessary to support a cloud tier in a mature storage product, in this paper, we focus on innovations needed to support key functionality for customers. Consider typical customer interactions: First, a customer determines which files to migrate to the cloud by estimating how much space will be freed on the on-premises Data Domain appliance. Second, a customer transfers selected files to the cloud and later restores files back. Finally, a customer deletes a file in the cloud when its retention period has expired. Each of these operations requires significant architectural changes and new algorithms to address both the impact of deduplicated storage and the latency and expense of cloud object storage. We also present analysis from deployed cloud tier systems. As an example, some customers have moved more than 20PB of logical data to the cloud tier and achieved a total compression factor (deduplication * local compression) of 40× or more, resulting in millions of dollars of cost savings.

1 Introduction

Today many customers want options to migrate portions of their data to cloud storage. Object storage in public and private clouds provides cost-effective, on-demand, always available storage. Data protection is a key requirement, and Data Domain [32] has traditionally served as an on-premises data protection product, holding backups of customers' primary data. Data Domain added a deduplicated *cloud tier* to its data protection appliances. Our deduplication system consists of an *active tier* where customers backup their primary data (typically retained for 30-90 days) and a cloud tier where selected backups are transitioned to cloud storage and retained long term (1-7 years). Our recent cloud tier product is currently being used by hundreds of customers.

Adding a cloud tier to a mature storage appliance involved numerous architectural changes to support local and remote storage tiers. We present some of the most novel improvements necessary to support basic capabilities required by customers. First a customer wishes to free up space on their active tier by migrating files to the cloud and wishes to determine how much space will be saved (Section 4). While this is straightforward to calculate in traditional storage systems, deduplication complicates the process because the files may have content that overlaps with other files that will remain on the active tier. We present a new algorithm to estimate the amount of space unique to a set of files. This algorithm builds upon a previous technique using perfect hashes and sequential storage scans [7] for memory and I/O efficiency.

Once a customer selects files for migration to the cloud tier, we wish to transfer the unique content to the cloud tier to preserve the benefits of deduplication during transfer. When a large quantity of data is being initially transferred, we developed a bulk seeding algorithm that also uses perfect hashes to select the set of chunks to transfer (Section 5). For ongoing transfers, we developed a file migration algorithm that leverages metadata that is stored locally to accelerate the deduplication process and avoids the latency of accesses to cloud object storage (Section 6.1). We then describe how files can be efficiently restored to the active tier.

Finally, as a customer deletes files from a cloud tier, unreferenced chunks must be removed to free space and reduce storage costs (Section 7.2). While garbage collection for the active tier has been described [11], we updated the cloud tier version to handle the latency and financial cost of reading data from the cloud back to the on-premises appliance.

From experience with a deployed cloud tier, we have learned lessons about sizing objects stored in private and public object storage systems and trade-offs of performance and cost. After analyzing deployed systems, we found that customers achieve a range of deduplication ratios. Our customers achieved an active tier deduplication ranging between 1× and 848× and cloud tier deduplication ranging between 1× and 66×. The space savings result in cost savings and one customer saved as much as $10 million (Section 8).

Our largest system has a single 1PB active tier and two 1PB cloud units within a single cloud tier. This is the physical capacity before the benefits of deduplication and compression. A cloud unit is a single deduplication domain. Each cloud unit has its own metadata, data and fingerprint

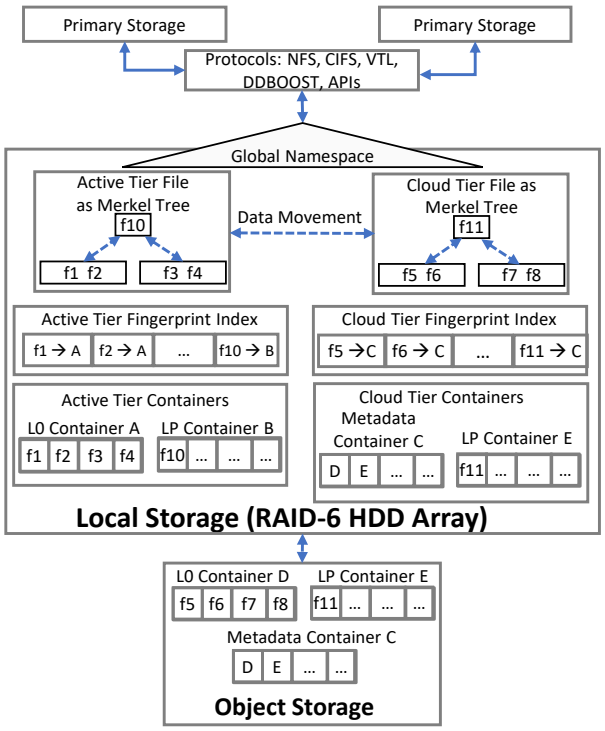

Figure 1: Data Domain with active and cloud tiers

index. For simplicity, we will use *active tier* and *cloud tier* as the terminology when referring to the active and cloud components of our backup appliance. Our cloud tier is designed to work with object storage that is either on-premises or in a public cloud, and we use the terms *object storage* and *cloud storage* interchangeably.

The vast majority of the technology described in this paper is publicly available and used by our customers. The exception is an experimental improvement to our garbage collection algorithm described in Section 7.2 based on microservices for public cloud providers. We believe the following contributions, except the performance results, are applicable to other deduplicated storage systems adding a cloud tier.

1. An architecture for a deduplicated cloud storage tier
2. A space estimation algorithm for files within deduplicated storage
3. Algorithms to seed a cloud tier or perform ongoing file migration
4. A garbage collection algorithm designed for cloud storage properties
5. An evaluation of how customers are using the cloud tier
6. Performance results on internal systems

2 Architecture

We updated our deduplicated storage architecture to support both a local, active tier and a remote, cloud tier. We first review the active tier architecture and then describe changes for a cloud tier as shown in Figure 1.

2.1 Active Tier Architecture

A file in both, active and cloud tiers, is represented by a Merkle tree with user data as variable sized chunks at the bottom level of the tree, referred to as *L0 chunks*. The SHA1 fingerprints of those chunks are grouped together at the next higher level of the tree to form chunks, referred to as L1 chunks. SHA1 fingerprints of L1 chunks are grouped together as L2 chunks, and this continues up to L6 which represents the entire file. The top chunk of the tree is always an L6 chunk, even though it may refer to chunks in a lower numbered level. We refer to chunks above L0 as LP chunks. The L6 chunk of every file is stored in a namespace which is represented as a B+ Tree [11].

Deduplication happens when different files refer to the same L0 and LP chunks. As an example, if two files are exactly the same, they would have the same L6 fingerprint. But if two files only partially overlap in content, then some branches of the tree will be identical (LP and L0 chunks), while other branches will have different fingerprints. Multiple L0 chunks are compressed into 64K-128K sized compression regions, while LP chunks are not compressed as SHA1 fingerprints are quite random and do not compress. If encryption is enabled, the compression regions are also encrypted before they are written to storage containers.

A Data Domain appliance tends to be utilized by a single customer, who typically selects a single encryption key for all of the data. If multiple keys are selected, customers accept a potential loss in cross-dataset deduplication. We have not found customer demand for convergent encryption or stronger encryption requirements for cloud storage than on-premises storage.

All chunks (L0s and LPs) are written into storage containers. On the active tier, containers are 4.5MB in size, while the container size on the cloud tier varies with properties of each cloud storage system (Section 2.3). We segregate the L0 and LP chunks into separate containers, which we refer to as *L0-Containers* and *LP-Containers*, respectively. Creating separate LP-Containers supports various operations like garbage collection that need to process LP chunks. Segregating L0 chunks from LP chunks also ensures that the locality of L0 chunks is preserved which results in better read performance. Both types of containers consist of a data section with the chunks and a metadata section with fingerprints of the chunks. During deduplication, container metadata sections are loaded into a memory cache. This loads 1,000 or more fingerprints into memory, helps to accelerate deduplication, and reduces fingerprint index accesses from disk. During reads, container metadata sections are loaded into the same memory cache and this avoids having to read all subsequent fingerprints in the container from disk [32].

2.2 Cloud Tier Architecture

With the introduction of cloud tier, there is a single namespace, referred to as global namespace. The global namespace spans both, active tier and cloud tier files. The Merkle trees of files in the cloud tier are stored on the local storage of the Data Domain system to facilitate high-speed access of those files. The global namespace which contains the L6 chunks of every file is periodically written to the cloud tier. In this design, files exist in one tier or the other based on a customer's policy. In some cases, customers use the cloud tier as extra capacity, while other customers use it for long term archival of selected data.

For the cloud tier, we introduced a third type of container, which we refer to as *Metadata-Container*. Metadata-Containers store the metadata sections from multiple L0 and LP-Containers. Since the metadata sections of containers are read during deduplication and garbage collection, and require quick access, Metadata-Containers are stored on the local storage as well as in cloud storage.

The SHA1 fingerprints of chunks are stored in an on-disk fingerprint index which consists of a mapping of fingerprint to container number. To avoid these writes and reads to the cloud, the cloud tier fingerprint index is stored on the local storage of the Data Domain system. In the active tier, the fingerprint index contains fingerprint to L0 or LP-Container number mappings, while the cloud tier index contains fingerprint to Metadata-Container number mappings. This is an optimization to load fingerprints from a local Metadata-Container instead of a remote L0 or LP-Container.

Figure 1 shows an active tier file that contains L0 chunks with fingerprints f1, f2, f3, and f4. The L0 chunk fingerprints are stored in an LP chunk with fingerprint f10. The L0 chunks and their fingerprints are stored in an L0-Container A, while the LP chunks and their fingerprints are stored in an LP-Container B. The active tier fingerprint index contains mappings for all the L0 and LP chunks' fingerprints. Figure 1 also shows a cloud tier file. The global namespace contains the location of this file. The file contains L0 chunks with fingerprints f5, f6, f7, and f8. The L0 chunk fingerprints are stored in an LP chunk with fingerprint f11. The L0 chunks and their fingerprints are stored in an L0-Container D, while the LP chunks and their fingerprints are stored in an LP-Container E. L0-Container D is written to object storage in the cloud, while LP-Container E is written to both, object storage in the cloud and local storage. The cloud tier fingerprint index contains mappings for all the L0 and LP chunks' fingerprints, and is stored on local storage.

As explained above, we store critical cloud tier metadata on the local storage of the Data Domain system to improve performance and reduce cost. The majority of this metadata, including the global namespace, is also mirrored to the cloud in order to provide a disaster recovery (DR) functionality. Disaster recovery is needed if a Data Domain system with a cloud tier was lost in a disaster. Such a disaster can result in the loss of active tier and the local cloud tier storage, where cloud tier metadata resides. Disaster recovery is the main reason why the active tier and cloud tier have different deduplication domains. If an active tier is lost, the backup copies migrated to object storage can be recovered.

The DR procedure to recover the cloud tier includes procuring a replacement Data Domain system and initiating an automated recovery process, which involves: creating a new cloud unit, copying the metadata (LP and Metadata-Containers) to the local storage for the cloud tier, rebuilding the cloud tier fingerprint index, and recovering the global namespace. Note that we only recover the metadata from the cloud storage. The data continues to reside in cloud storage, and it is not copied to the Data Domain system. After the DR procedure is completed, the cloud tier is accessible from the new Data Domain system.

To summarize our new architecture, we support an active tier and a cloud tier. Both tiers offer the benefits of deduplication, however each tier is a separate deduplication domain. Chunks are not shared between the active tier and the cloud tier. Note that if the underlying object storage provides deduplication, it would be ineffective since our deduplication algorithm removes the majority of duplicates at 8KB granularity. A global namespace maintains the location of both, active tier and cloud tier files. Each tier has its own metadata, data, and fingerprint index. To eliminate the costly reads and writes from/to the cloud, we copy key cloud tier data structures on the local storage of the Data Domain system. The data structures stored locally are the Metadata-Containers, the LP-Containers, the Merkle trees, and the cloud tier fingerprint index. The Metadata-Containers and LP-Containers are also mirrored to the cloud to facilitate disaster recovery.

2.3 Object Sizes for Cloud Tier

The cloud tier architecture we have chosen allows us to select an optimum object size because we write L0-Containers as individual objects and we have the ability to control the container size, up to 4.5MB. We are not able to write containers larger than 4.5MB without significant changes to our implementation. In the case of cloud tier, we use the terms objects and containers interchangably.

We started with 64KB objects, but evolved to larger sizes in the range of 1-4MB for several reasons. Larger objects result in less metadata overhead at the cloud storage provider because they store per-object metadata. Larger objects also decrease transaction costs as cloud storage providers charge per-object transaction costs. We have also discovered that larger objects perform better. Using an internal tool, we experimented with object sizes ranging between 64K and 4MB. On private cloud storage such as ECS [8], we saw a 6x improvement with 4MB objects compared to 64KB objects. On public cloud storage such as Amazon S3 [3], we saw a 2x

improvement with 4MB objects compared to 64KB objects. Even though the performance does not improve much after 1MB, a higher object size is better as it reduces the cost of transfer. Ultimately, writing larger-sized objects is a better choice for our cloud tier solution, though the exact size we choose varies with properties of each cloud storage system. In some cases certain object sizes align better with the provider's block size. For different providers, we choose different object sizes ranging from 1MB to 4MB.

3 Background on Perfect Hashing and Physical Scanning for the Cloud Tier

In addition to the architecture changes described in Section 2.2, we introduced several new algorithms specific to cloud tier. Since these algorithms utilize the perfect hashing technique from Botelho et al. [7] and the physical scan technique of Douglis et al. [11] as building blocks, we briefly review those works. Perfect hashing and physical scanning provide the basis for building the following cloud tier algorithms: space estimation (Section 4), seeding (Section 5), and cloud tier garbage collection (Section 7.2). Adapting perfect hashing and physical scanning to these cloud tier algorithms was mostly an engineering effort, and the novelty is specific to solving the challenges of the cloud tier algorithms and not necessarily the underlying techniques used.

3.1 Perfect Hashing

For algorithms described below, we need to perform a membership query for our fingerprints. Perfect hashing is a technique that helps us to perform this membership query by representing a fixed key set. We use a *perfect hash* vector which consists of a perfect hash function and a bit vector [7, 11]. A perfect hash function is a collision-free mapping which maps a key to a unique position in a bit vector. To generate the hash functions, a static set of fingerprints under consideration is used to generate a 1:1 mapping of fingerprint to a unique position in the bit vector. The function building process involves hashing the fingerprints into buckets where multiple fingerprints map to one bucket. Then for the fingerprint set in each bucket, we build a perfect hash function. By dividing the fingerprints into these buckets, we can build functions for each bucket in parallel. Once we obtain the function for each bucket, we store the function in a compact way [5, 6, 9]. Without the compactness of the perfect hashing representation, we would not have sufficient memory to reference all fingerprints in the system.

Building the perfect hash functions is quite efficient. For example, on a full 1PB system, we can build the perfect hash functions for 256 billion fingerprints in less than 3 hours with about 2.8 bits needed per fingerprint. The bit value in the perfect hash vector is used to record membership, such as chunk liveness for garbage collection.

3.2 Physical Scanning

As files are deduplicated, new LP chunks are written which refer to lower level LP and L0 chunks. Say two L2's written by two different files refer to the same L1, then these L2s most likely will get written to different containers. Hence these LP chunks get fragmented over time. For algorithms described in the sections below (e.g. garbage collection), we need to walk the LP chunks of all or most of the files in the system.

One way to walk the Merkle trees of LP chunks is to do it in a depth first manner. For every file, walk from L6 chunk, to L5, L4, L3, L2 and down to L1 chunks and get the L0 refererences. There are two problems with this traversal. First is that if two files point to the same LP chunk, then by doing file by file walk in depth-first manner, we will enumerate the same LP twice. The second problem is that since these LP chunks can be in different containers, loading these LP chunks will result in doing a random lookup to first get the location of LP chunk from the fingerprint index and second to read the LP chunk. Over time as deduplication increases, the same LP and L0 chunks get referenced multiple times and the LP fragmentation worsens.

Hence, instead of doing an expensive depth-first traversal, enumeration is done in a breadth first manner. By keeping track of LP chunks we have already enumerated (using a perfect hash vector), we avoid enumerating the same chunk twice. To reduce the random lookups, we first segregated LP and L0 chunks in different containers. By doing this, we converted random lookup for every LP to a random lookup for a group of LPs present in same compression region of the container. Here is a summary of the steps we follow to perform a physical scan of the file system:

Analysis Step: We walk the on-disk fingerprint index to create three perfect hash vectors as described in Section 3.1. Two of the perfect hash vectors are called the *walk vector* and *read vector*, respectively, and are used to assist in the breadth-first walk of the Merkle trees of all the files. These two vectors are only built for LP chunks. The third vector is called the *fingerprint vector*, and is used to record the liveness of a fingerprint. The fingerprint vector has bits for LP and L0 chunks. 97% of chunks are L0s, so the fingerprint vector is the largest.

Enumeration Step: We perform a number of sequential scans of containers to find chunks at specific levels in the Merkle trees. We first walk the namespace and mark all L6 fingerprints in the walk and fingerprint vectors. The top chunk is always a L6 chunk, which may refer to any lower-numbered chunk. Our system has an in-memory structure that records which LP types (L6, L5, ... L0) exist in each container, so we can specifically scan containers with L6 chunks. Figure 2 shows the next two steps of the enumeration process along with how the perfect hash vectors are used. Blue indicates the state of perfect hash vectors in the

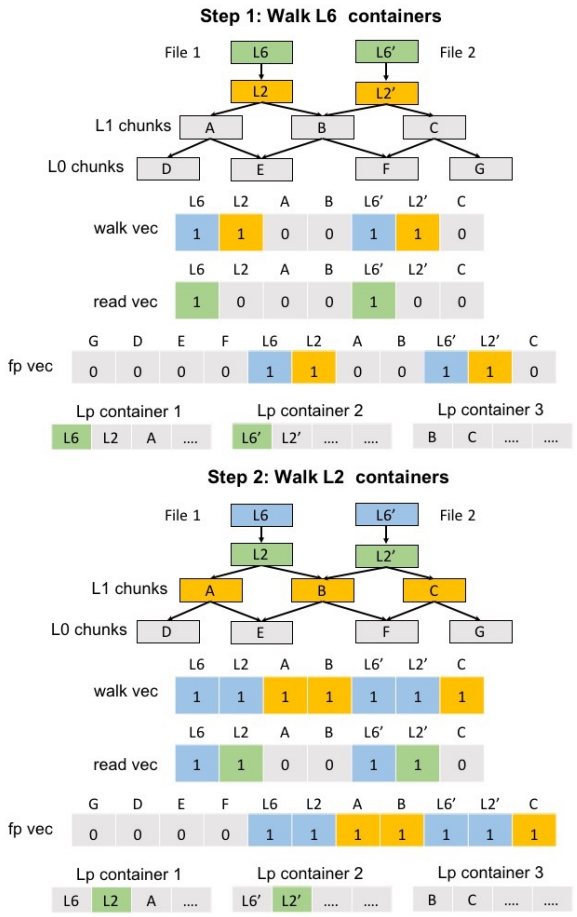

Figure 2: Physical Enumeration process

previous step, green indicates that we are reading containers and setting the read vector and yellow indicates that we are setting the walk and fingerprint vectors and gray indicates that we are yet to process those fingerprints. In step 1, we walk the container set reading L6 chunks and for any L6 fingerprints marked in the walk vector, we mark that L6 in the read vector (green). In this example, the L6 chunks reference L2 fingerprints, so we mark the L2 fingerprints in the walk (yellow) and fingerprint vectors (yellow). In step 2, we walk the container set again reading L2 chunks and for any L2 fingeprints which are marked in walk vector, we mark those L2 fingerprints in read vector (green). We read the L1 fingerprints from the marked L2 chunks and mark those L1 fingerprints in the walk (yellow) and fingerprint (yellow) vectors. For deeper trees, we repeat the steps of reading a level, marking fingerprints in a walk vector and fingerprint vector, and then reading the lower level and marking fingerprints in the walk vector. Finally, in the L1 container set walk, the L1 chunks have a list of L0 fingerprints, which we mark in the fingerprint vector. Parallel I/O is leveraged to read containers from the RAID array, and multiple threads are used for marking the chunks in the walk, read, and fingerprint vectors.

4 Estimate Freeable Space

Deduplication creates a new challenge for customers interested in reducing their active tier footprint. In traditional storage, transferring 10GB of files would reduce the active tier by 10GB. With deduplicated storage, less space may be freed because of content overlap with files that remain on the active tier. So, a customer may end up paying for on-premises capacity as well as object storage capacity despite their intention.

Such customers need a way to evaluate how much space would be freed on the active tier by moving files to the cloud tier. Variants of this problem have been considered for directing content to storage nodes [10, 12, 20] to maximize deduplication and to evaluate the unique space referenced by volumes of block storage [14]. Our system does not maintain reference counts due to the difficulty of maintaining accurate counts under complex error cases, so we implemented an algorithm to calculate the space that would be saved on the active tier if selected files were migrated to the cloud.

Using the perfect hashing (Section 3.1) and physical scanning (Section 3.2) techniques, we walk the files selected for migration in a breadth-first manner through the Merkle tree and mark the chunks in an in-memory perfect hash vector. Then, we walk all the remaining files (those not selected for migration) and repeat the breadth-first traversal again. In the second traversal, we unmark the chunks referenced by these remaining files. After this traversal is complete, the chunks which are still marked in the perfect hash vector are the chunks which are uniquely referenced by files selected for migration to the cloud tier. We perform a walk of the containers and sum up the chunk sizes for any chunks marked in the perfect hash vector.

This gives us an exact count of the bytes that would be freed. It becomes an estimate when new files are written to the active tier after constructing the perfect hash vector and such files deduplicate with selected chunks. In such cases, our algorithm overestimates the number of bytes that would be freed due to migration.

Since space estimation uses perfect hashing, the estimation cannot be done until the perfect hash functions are generated. As discussed in Section 3.1, this process takes nearly 3 hours for 1 PB of physical data. Also, this process gives only a point in time space estimate. As new data gets written to active tier, it can deduplicate against the chunks that will be moved to the cloud storage. As a result the point in time estimate becomes stale. Our customers can run this tool periodically to get the updated estimate. In practice, this has not been an issue.

5 Seeding

When customers started using cloud tier, they faced new challenges with data migration to the cloud. Some customers

had hundreds of TBs of data on the active tier that they wanted to migrate to the cloud tier. Being a deduplication system, migration of large amount of data to the cloud suffered from the same challenges as described in Section 3.2. Hence, we implemented a seeding algorithm based on the perfect hashing (Section 3.1) and physical scanning (Section 3.2) techniques.

We first build an in-memory perfect hash vector for fingerprints in the active tier fingerprint index. Then, for all the files that need to be transferred to the cloud tier, we traverse the Merkle trees in the same breadth first manner and mark the corresponding chunks live in the perfect hash vector. We then walk the containers on the active tier, pick the chunks which are marked live in the perfect hash vector, pack them in destination containers, and write them to cloud tier. This process generates both data containers (L0) and containers for metadata (LP and Metadata-Containers). Once all these containers are written, we update the namespace to indicate that all these files are now located in the cloud tier.

Seeding needs to be resumable because the memory requirements for seeding and GC are high and memory is shared between these processes. Cloud GC and seeding share memory, but until seeding is complete, there is no need to run cloud GC since the cloud tier is nearly empty. Active GC and seeding may need to run at the same time, especially since seeding may take weeks to transfer data depending on WAN speed. To make it resumable, after we mark the chunks that need to be transferred in the perfect hash vector, we persist the perfect hash vector (function and bit vector) to disk and then start active GC. Once active GC finishes, we load the perfect hash vector and resume writing to the cloud.

Seeding guarantees that all the L0 and LP chunks are transferred to the cloud tier before the file's location is changed in the namespace. It uses the perfect hash vector to guarantee that property. As we transfer chunks to the cloud tier, we change the membership bit in the perfect hash vector from 1 to 0. At the end of seeding, all the bits in the perfect hash vector should be 0 to guarantee that all necessary chunks are transferred.

There is an important caveat with seeding. The chunks written to the cloud tier do not pass through the deduplication process. So, seeding is only used to transfer large amounts of data for the first time when a customer buys a cloud tier license. We cannot use the typical fingerprint caching and prefetching approach for deduplication because the seeding process works in physical disk order instead of the logical order of chunks in a file, so caching is ineffective. Another approach is to consider generating a perfect hash vector for data already stored in the cloud tier, but this would double the memory requirements since we would need to generate perfect hash vectors for both the active and cloud tiers. We already use a large perfect hash vector and cloud tier had to be supported on existing active tier systems in the field, so adding more memory for this is not a practical solution.

6 File Migration and Restore

6.1 File Migration

Unlike seeding which is a one time process to transfer a large amount of data from the active tier to a nearly empty cloud tier, file migration is designed to transfer a few files incrementally. File migration reduces the amount of data transferred to the cloud tier by performing a deduplication process relative to chunks already present in the cloud tier.

File migration starts in the active tier by traversing the Merkle trees for selected files in a depth-first search manner. This traversal is performed in parallel for a number of selected files. The traversal ends when we reach the desired L1 chunks, which contain L0 fingerprints. The fingerprints are checked against the cloud tier using the same container metadata prefetching technique used during deduplication to see if identical fingerprints are already present in the cloud tier. If the fingerprints are not present in the cloud tier, then the corresponding chunks are read from the active tier and packed to form new L0-Containers. This process generates both data containers (L0) and containers for metadata (LP and Metadata-Containers). After Metadata-Containers are written, all contained fingerprints are added to the cloud tier fingerprint index. Finally, the namespace is updated to indicate that the file is located in the cloud tier.

To compare seeding and file migration, seeding is designed for bulk transfers such as when first moving files from a full active tier to nearly empty cloud tier. Seeding has the overhead of generating the perfect hash function, which is nearly 3 hours per 1PB of physical capacity. This is acceptable relative to the many days or weeks possibly required to transfer a large dataset to the cloud. Seeding has the advantage of physical enumeration, which uses sequential I/O instead of random I/O. On the other hand, when transferring a small number of files, it is more efficient to perform the random I/O to find the needed chunks for the files than generate the perfect hash function. Newer backup files on the active tier also tend to have better physical locality as our deduplication engine explicitly writes duplicates to keep locality high [2, 25]. Seeding and file migration are experimentally compared in Section 8.

6.2 File Restore

Restoring a file back from the cloud tier involves the reverse process of reading the Merkle trees to the L1 chunks in a depth-first manner. The L1 chunks are read from local storage since the LP-Containers are stored locally. The L0 fingerprints are checked against the active tier fingerprint index, and if present, do not need to be read from cloud tier. For new L0 fingerprints (not present in the active tier), we get the Metadata-Container number from the cloud tier fingerprint index, read the Metadata-Container to get the L0-

Container number, read the L0-Container from object storage, and write the relevant L0 chunks to L0-Containers in the active tier. As L0-Containers are being written to active tier, new Merkle trees for the active tier are formed in a bottom-up manner. This process generates new LP chunks which are written to LP-Containers in the active tier, and the namespace is updated.

To summarize, while performing file migration to the cloud tier, we deduplicate against chunks in the cloud tier, and while restoring files from the cloud tier, we deduplicate against chunks in the active tier.

7 Garbage Collection (GC)

When customers expire backups, some chunks become unreferenced. The Data Domain filesystem is log-structured and garbage collection is responsible for cleaning the unreferenced data asynchronously. Our garbage collection is mark and sweep based and described in our previous work [11]. To briefly summarize the process, we use the physical scanning technique described in Section 3.2 to mark chunks live (referenced from live files) in the perfect hash vector. Then, we perform the sweep operation. Active tier GC and Cloud tier GC differ in the sweep process, and we describe the differences here. Most of our customers run GC on the active tier weekly according to a default schedule, whereas cloud tier garbage collection varies from once every two weeks to once every month to never.

7.1 Active Tier Garbage Collection

After marking chunks live in the perfect hash vector, the sweep process walks the container set to copy live chunks from old containers into newer containers while deleting the old containers. This sweep process is also called copy forward. This process focuses on a subset of the container set which will give us the maximum space back.

During this process we first read the source containers from disk. Then, we check the fingerprints from the metadata sections of these containers against the perfect hash vector to determine which L0 chunks are live. Finally, we decrypt (if encrypted) and uncompress the compression regions inside the source containers, encrypt and compress the live chunks into new compression regions, and pack them into destination containers which are written to disk. The copy forward process for LP Containers does not perform decompression/-compression since LP chunks are not compressed.

7.2 Cloud Tier Garbage Collection

As data is written to the cloud tier, space usage and costs grow. Similar to the active tier, when customers expire files in the cloud, GC needs to clean unreferenced chunks on the cloud tier. The challenge for cloud GC is that L0 containers are not local and reading them from the cloud is expensive. It is also slow in terms of performance as we have to read the container objects over WAN.

From our experience with active tier, we know that a single cycle of GC copies forward 15% of the containers on average, where each container has an average of 50% live data. Hence, for 1PB of physical capacity in a cloud tier, we need to read 150TB of partially-live containers and write 75TB of newly-formed containers, increasing data transfer costs and transactional costs. As an example, based on AWS pricing, we calculated the cost of copy forward for 150TB of data to be nearly $14,000 per GC cycle. The major contributor to this cost is egress data transfer cost, so we needed a way to do garbage collection without reading the L0-Containers from object storage and writing new L0-Containers to object storage. To address this, we defined an API to perform copy forward inside the cloud provider's infrastructure. ECS and VirtuStream cloud providers have implemented this API. The API takes a list of source container objects, a list of offset ranges inside of those container objects, and a destination container object. The offset ranges correspond to live compression regions within the source container objects. Upon receiving the API, the cloud provider copies the live offset ranges from the source container object to a new, destination container object.

In order to perform the chunk level copy forward done by active tier GC, the compression regions need to be decrypted and uncompressed, and the live chunks need to be copied forward to form new compression regions inside new containers. Doing all of this in the cloud provider's infrastructure poses a challenge as we need to expose our container format, compression libraries, and encryption keys to the cloud provider. To address this, the new API we have implemented does not decompress/compress or decrypt/encrypt compression regions. It performs copy forward at the compression region level. We only delete a compression region in a cloud container when it is completely unreferenced. This approach is different from active tier GC where we delete individual chunks. The advantage of treating an entire compression region as live or dead is that the service running inside the cloud does not need to understand the container format or to compress/uncompress or decrypt/encrypt the compression regions. The service just reads offset ranges of where the live compression regions reside in existing container objects and writes a new destination container object. The disadvantage is that we won't delete a compression region until all the chunks inside the compression region are unreferenced. This can reduce our cleaning efficiency. In Section 8.1.2, we present field analysis of how live and dead data is distributed in compression regions and how much cleaning efficiency is lost due to compression region based cleaning instead of chunk based cleaning.

The cloud GC copy forward process works as follows. Once the live chunks have been identified in the per-

fect hash vector using the physical scanning technique described in Section 3.2, cloud tier GC performs a copy forward process of the Metadata-Containers. This process copies the live chunks' metadata from an existing Metadata-Container to a new Metadata-Container, and deletes the old Metadata-Container. The copy forward process of Metadata-Containers occurs on the local storage as Metadata-Containers are stored locally. As part of this process, we create a recipe containing the old L0-Container, the offset ranges of the live compression regions in that L0-Container, and a destination L0-Container which is being generated. Next, we send this recipe to the cloud provider in the form of the new API, and the cloud provider performs the copy forward of compression regions within their infrastructure. We then delete the old container objects.

For public cloud storage like AWS, Azure, and Google Cloud, such an API does not yet exist, so we have created an experimental version (not yet available to our customers) using a microservice that can be deployed and run inside the public cloud provider environment. Our plan is to work with cloud providers to either use a custom API for copy forward or our microservice. The results from the API and microservices based approaches are presented in Section 8.

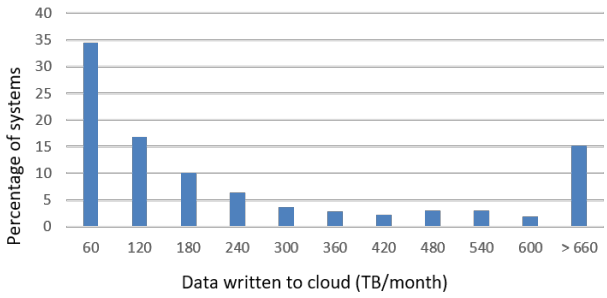

Figure 3: Distribution of TB/month moved to the cloud

Figure 4: Data moved to the cloud versus total compression

8 Evaluation

We divide our evaluation into results from deployed systems and results from internal experiments.

8.1 Deployed Systems Evaluation

In this section, we show how our customers are using cloud tier in terms of data written, deduplication, and data deleted. We do not present results for space estimation and seeding in this section as these were recently released and the sample set is statistically small. Those techniques are evaluated on internal systems. We considered results from hundreds of deployed systems, filtered out systems with less than 1TB (post dedup/compression) of data in the cloud tier and inconsistent reports, and randomly selected 200 systems for analysis. Across systems, the age of the cloud tier varied from a few months to over two years.

Figure 3 shows monthly cumulative bytes (before deduplication and compression) sent from active tier to cloud tier for our 200 systems in the last three months. On one end, in 35% of the cases, less than 60TB (before dedup/compression) per month has been moved to cloud tier. On the other end, in 15% of the cases, over 660TB per month was sent to the cloud tier.

Figure 4 shows a scatter plot comparing the amount of data (before deduplication and compression) the deployed systems have moved to cloud tier and the total compression they have achieved. Total compression ratio is defined as deduplication ratio * local compression ratio. The general trend is that as more data is moved to the cloud tier, the more total compression is achieved because the new data deduplicates with older data. But there are some cases where even after moving large amounts of data to the cloud tier, the total compression is low. We further analyzed those systems and discovered that those customers are selecting and moving datasets with low total compression to the cloud tier to reduce their on-premises cost. Interestingly, we also found a few systems that have moved 20PB or more to the cloud tier and achieved a total compression factor of 40× or more. One system in particular achieved 66× total compression after moving 40PB of logical data to the cloud tier, resulting in substantial space and cost savings.

To understand the cost savings seen by our customers, we calculated how much money our customers are saving due to deduplication and compression. Even though customers are using various cloud providers, for simplicity, we assume Amazon S3 cost metrics for transaction and storage costs. We calculated total storage costs by calculating a monthly storage cost and accumulating it for all the months. We then calculated the transaction costs based on the total number of transactions performed and the cost of each transaction. Next, we added the storage and transaction costs based on bytes written before and after deduplication and compression, and we calculated the difference of the two. The histogram in Figure 5 shows this difference which represents the cumulative cost savings due to deduplication and compression in thousands of dollars on a log scale. Some of our customers saved nearly $10 million due to deduplication and

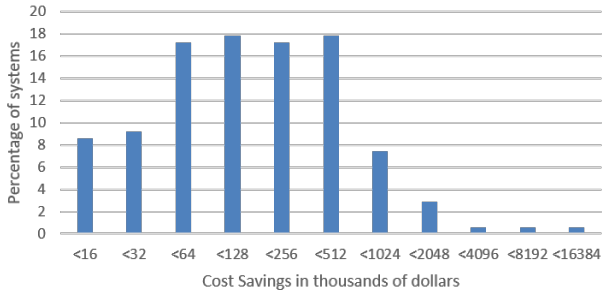

Figure 5: Histogram of cumulative cost savings in thousands of dollars due to deduplication and compression

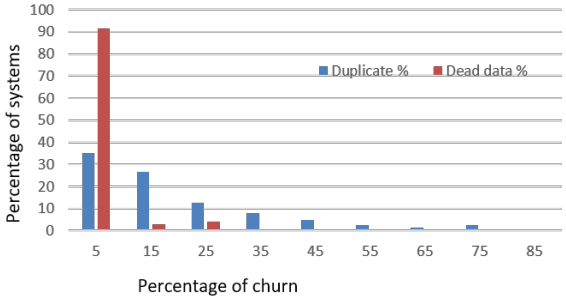

Figure 6: Comparison of physical churn due to file deletes and duplicates

compression. As customers retain the cloud tier data for a longer time period or write more data, their savings due to deduplication and compression will further increase.

8.1.1 Field GC Analysis

To understand how customers are deleting data and how much churn is really generated, we looked at customers who ran GC at-least once (40% of our selected 200 systems). As customers delete files in the cloud tier, data becomes unreferenced and needs to be cleaned. As mentioned in Section 7.2, our system writes some duplicates to improve restore performance. In turn, our GC algorithm retains the most recent version of a duplicate (written to the newest container) and removes older duplicates to reclaim space. Hence there are two types of chunks that can be freed from our system: unreferenced chunks due to file deletions and duplicate chunks. Figure 6 presents the percentage of physical churn (bytes deleted within a time period) that is dead due to file deletions and due to duplicates relative to total capacity of the system. Results from the last three months are included for systems that have run cloud GC at least once.

The churn due to file deletions is relatively low-90% of the systems had less than 5% of space reclaimed for dead chunks, because deduplication creates many references to chunks and because customers tend to retain their cloud data for long periods. There are some cases where the churn due to file deletions is over 25%, suggesting some customers

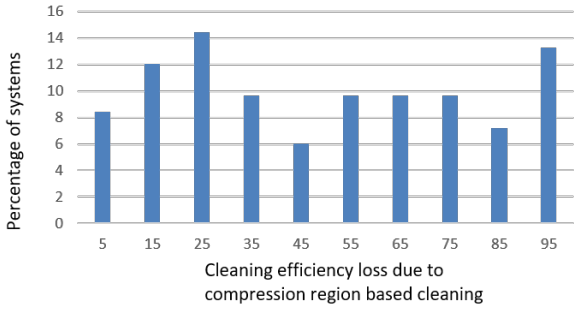

Figure 7: Percentage of cleaning efficiency loss

delete a large fraction of their cloud tier periodically. In the steady state, most customers continue to delete the oldest backups as newer backups are stored. This graph also shows that while most of the systems have less than 30% of their churn as duplicates, in some cases the duplicates are very high (up to 75%). We looked at those systems and found that some of those customers have never run GC or have not run GC recently, so duplicates have not yet been removed. In contrast, active tier GC runs each week, so duplicates do not accumulate. We have learned that retention policies differ between active and cloud data, and we have provided an option to customers to control the amount of duplicates they want to write to cloud tier in case they infrequently run GC.

8.1.2 Cleaning Efficiency Loss due to Compression Region Cleaning

We analyzed the 200 systems to understand how much cleaning efficiency we lose with compression region-based cleaning for the cloud tier relative to possibly running chunk-level cleaning. Figure 7 shows the percentage of bytes that cannot be deleted by compression-region level cleaning but could have been deleted by chunk-level cleaning. As we can see, the cleaning efficiency loss is almost 100% in some cases. In such cases, compression-region level cleaning won't delete anything. We looked at some of these systems closely and found that this happens because the churn (bytes deleted within a time period) in the cloud tier is low. Even though there are a lot of duplicates that can be removed, these duplicates reside in the same compression regions as chunks that are still live, and this prevents us from deleting the compression region. In contrast, chunk-level cleaning is able to delete the duplicates while keeping only the live chunks. Further analysis of this observation is needed as the frequency and pattern of deletions and deduplication can result in different amounts of cleaning efficiency loss. In cases where we are not able to free up any space using compression-region level cleaning, we offer a chunk-level cleaning option, which performs the traditional algorithm of reading and writing container objects over the WAN. We also plan to augment our API and microservice based cleaning services to perform chunk level cleaning in the future.

Hardware	DD-Mid	DD-High
Memory	192GB	384GB
CPU(cores * GHz)	8 * 2.4GHz	24*2.5GHz
Active tier capacity	288TB	720TB
Cloud tier capacity	576TB	1440TB

Table 1: Data Domain hardware for experiments

8.1.3 Deployed Systems Summary

Here are findings from our cloud tier deployments:

1. Some customers are writing 500TB logically per month while others are writing 100TB or less.
2. Customer data is achieving a broad range of total compression ratios, from less than 4× to 100×, because customers are using a cloud tier in different ways. Some are writing highly redundant data to cloud tier (their total compression factor on the cloud and active tiers are both high). Such customers may accumulate more metadata than we originally anticipated and this can affect the runtime of GC and other algorithms. Other customers are writing non-redundant data to the cloud tier. It is likely that such customers are choosing low-deduplication data to migrate to the cloud tier. Other customers have not written much data to the cloud tier yet, so their total compression factor is lower.
3. Some customers have more data written to the cloud tier than to the active tier, so these customers are trying to reduce their on-premises storage cost.
4. Churn on the cloud tier (0-5% per month) is substantially lower than churn on the active tier (10% per week) because customers are not deleting much data. This finding is aligned with our expectation that cloud tier is used for long term retention.
5. Most customers are running cloud tier GC infrequently or not at all and have accumulated a high number of duplicates. Modifications to internal parameters can reduce the number of duplicates in these situations.

8.2 Experiments on Internal Systems

In this section, we focus on results from internal systems experiments. For all experiments, we used two cloud storage systems, Amazon S3 (public cloud) and ECS (private cloud). We used two Data Domain systems shown in Table 1, with the cloud tier feature as described in this paper. The two systems are representative of midrange (DD6800) and high-end (DD9300) products. We provide the size of the active tier and local storage for the cloud tier configuration.

8.2.1 Load Generator

To measure performance of our algorithms, we used an in-house synthetic load generator described previously [1, 11, 7]. A first generation backup is randomly generated, and

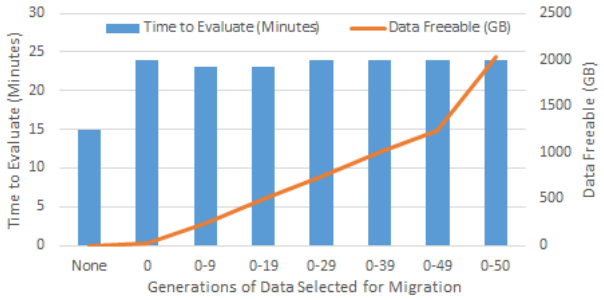

Figure 8: Space estimation performance

the following generations have deletions, shuffles, and additions. We typically write many streams in parallel, each stream consisting of generations of backups beginning from a unique seed. The change rate between two consecutive generations is 5%. Gen0 is the first generation of backups where we only get local compression (an average of 2×) and no deduplication. GenX is aged data where generations Gen0-Gen(X-1) are already written to the active or cloud tiers. Due to deduplication, only the new content in GenX is stored while the rest deduplicates. In the experiments below, we vary the initial backup size, number of generations, and number of parallel streams. Beyond 42 generations, the physical locality of data is degraded at a similar level as what many of our customers experience. In one experiment (Section 8.2.4), we generated 100 generations to explore the impact of extremely poor locality.

8.2.2 Freeable Space Estimation

To evaluate space estimation, we used the synthetic generator to create a data set, selected portions of the data set to potentially migrate, and ran the space estimation algorithm. We created a dataset using 96 parallel streams, each writing generations 0-50 of backups that average 24GB each for a total logical size of 120TB. Figure 8 shows the evaluation time and amount of space that can be freed as we vary the number of generations selected to potentially migrate. With no generations selected (`None`), the evaluation time is 15 minutes, and no space can be freed. As the number of generations selected increases up to including every generation (the rightmost bar), the evaluation time is consistently about 24 minutes while the amount of freeable space increases with the number of generations selected. When all generations are selected, the amount of freeable space jumps since chunks common to the highest generation can finally be freed. These results show that our space estimation algorithm has a run time based on the allocated space of the system. This is because space estimation does physical scans of metadata chunks followed by a container walk to calculate the estimated space that will be freed. The time for both physical scans and a container walk are a function of the physical space on the system.

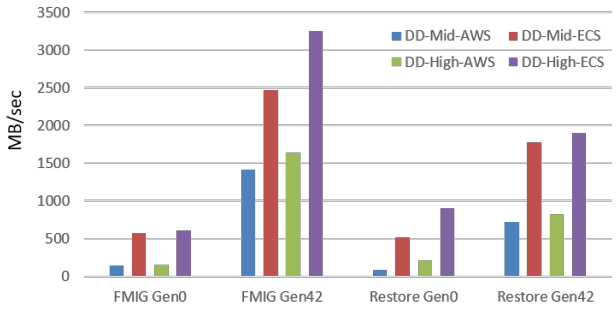

Figure 9: File Migration and Restore performance

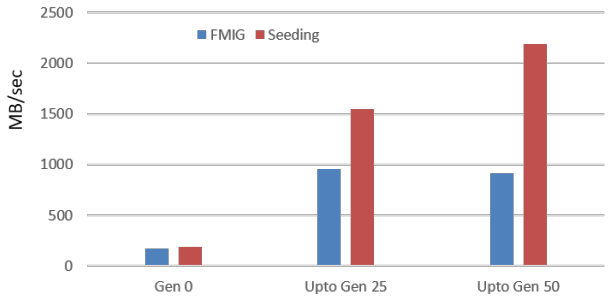

Figure 10: File Migration vs. Seeding performance

The duration of each run is only a few minutes because we wrote a small dataset instead of filling up a 1 PB system, which can take weeks. From experiences with active GC runs on a 1PB system (with the same underlying perfect hashing and physical scans), space estimation on a system of such capacity should finish within a couple of days.

8.2.3 File Migration and Restore from Cloud

Figure 9 shows the logical performance of file migration and restore on DD-Mid and DD-High using both AWS and ECS for object storage. We connected to AWS across the public Internet, while ECS was within our lab. We used Gen42 data to model aged data with high deduplication ratio. Gen42 file migration performance is higher than Gen0 file migration performance because of deduplication, as we only need to transfer the changed data to the cloud tier. In the case of Gen0 file migration, the performance for both hardware configurations is the same. This is because for Gen0, we are reading non-deduplicated data from the active tier and writing to the cloud tier, which is mainly gated by object storage latency and the hardware configuration does not have much impact. But in the case of Gen42, DD-High performs 16% better than DD-Mid on Amazon S3 and 31% better on ECS. This happens because Gen42 has highly deduplicated data compared to Gen0, and DD-High is able to sustain a higher throughput because it has higher parallelism. High object storage latencies continue to be a bottleneck, otherwise the performance difference between the systems would be higher. Gen42 restore from object storage is better than Gen0 restore because in case of Gen42 we are deduplicating against the previous generations of active tier. Also, the latest generation has better locality and hence better performance as GC tries to keep the latest copy of the chunk and hence over time the old generations get fragmented.

Restoring from object storage is typically slower than writing, because it involves reading data from different objects in object storage and writing to the active tier. For comparison, Gen42 write performance on the active tier with the same hardware is 3x better than writing to ECS and 2x better than restore performance from ECS. The major difference in performance is due to object storage latencies. On the active tier latencies of 10-50ms latencies are common, while on the cloud tier these latencies vary from 100 ms to 1 second for both public and private cloud vendors.

8.2.4 File Migration vs. Seeding Performance

Seeding does a bulk transfer of data in a breadth-first manner as opposed to the depth-first manner of file migration. To fairly compare the two algorithms, we did a transfer using both techniques. We ran three sets of tests, where we wrote Gen0, Gen0-50, and Gen0-100 to the active tier and then moved Gen0, Gen0-25, and Gen0-50 to the cloud tier, respectively, using both seeding and file migration. After every 5th generation written, we ran active tier GC and forced it to copy forward 30% of the containers to remove duplicates and degrade the physical locality of the data on the active tier. This simulates the scenario where customers have old, highly deduplicated data on the active tier and would like to move 50% of their oldest data to the cloud tier.

Figure 10 shows that seeding and file migration have similar performance for Gen0 because locality is high and seeding has the overhead of generating the perfect hash functions. As we transfer Gen0-25 and Gen0-50, seeding is faster than file migration by a 2x factor. This is because Gen0-50 have highly deduplicated data with degraded locality and the depth-first approach of file migration has to traverse the same containers repeatedly and incur random I/Os. In the case of seeding, the sequential scans during the breadth-first traversal compensates for the overhead of perfect hashing hence making the movement more efficient. This experiment was only performed on a DD-Mid system with Amazon S3 as the cloud storage. Similar to the Gen0 file migration performance discussed in Section 8.2.3, the hardware configuration does not have much impact because we are bottlenecked by the network throughput.

8.2.5 Garbage Collection Performance

Our analysis focuses on the copy forward process of cloud tier GC, as shown in Figure 11, since it is the only phase that differs from active tier GC. As described in Section 7.2, we developed a new API to perform copy forward for pri-

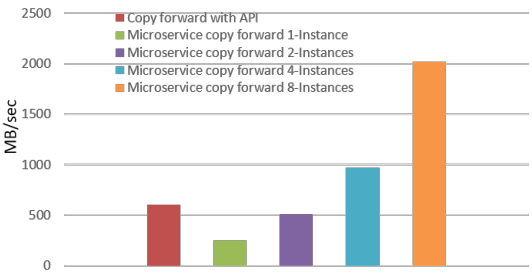

Figure 11: GC copy forward with different algorithms

vate cloud providers. Using this API, the system performs a copy forward of live compression regions (64-128KB) from existing container objects into new container objects without reading the data back to the Data Domain system to avoid transfer costs. Using the API, we are able to achieve a copy forward performance of 600MB/s.

For public cloud providers that do not provide a copy forward API, we developed an experimental microservice algorithm for garbage collection. The Data Domain system makes compression region cleaning decisions and passes a recipe to one or more GC microservice instances running in the cloud that perform the copy forward operations, similar to the functionality of the API. We increased the number of AWS t3.xlarge instances [4] to evaluate how well it scales. Aggregate performance scaled close to linearly, starting at 256 MB/s with one instance and increasing to 2,040 MB/s with eight instances. Even though microservice performance is better than API performance, we can use a microservice only in the case of a public provider where we can spin up compute to run the microservice. This might not be possible in private cloud environments.

9 Related Work

Deduplication is a well-studied field with multiple survey articles [21, 23, 31]. Deduplication is a key aspect of the Data Domain product to enable space savings and high performance for backups [32], and Data Domain has evolved with media and use case changes [1, 2].

There have been multiple papers characterizing backup data [15, 17, 19, 26, 30], and the terms backup and archive are often used interchangeable, so the previous analysis may have applications to archival system design. In our data analysis, customers have specifically decided to archive a subset of their backup data for longer term retention in the cloud.

Reading from deduplicated cloud storage can be slow, and several papers suggest ways to improve read performance, usually involving writing duplicates, caching, and prefetching techniques [18, 27]. Security implications of cloud storage have also been considered [16, 22, 24]. In contrast to these papers, we show how to evolve an existing deduplicated backup product to support a cloud tier.

The issue of deciding where to place large directories to maximize content overlap has been considered [10, 12]. Nagesh et al. [20] presented a technique to partition a collection of files by related content using an in-memory graph relationship on fingerprints. A current publication represents the content of storage volumes with sketches of sampled fingerprints to determine unique content for volumes [14]. In contrast, our technique can estimate the amount of space referenced from an arbitrary set of files selected by the user and scales to PB capacity.

Cumulus [28] provides backups to cloud storage by transferring file differences and storing files in large objects. BlueSky [29] presents a file system backed by cloud storage that uses a local cache for performance. Though neither incorporates deduplication, both projects describe garbage collection for large objects in the cloud as regions become unreferenced. Fu et al. [13] improve GC and restore performance in deduplicated storage by analyzing the history of container references during a backup. They rewrite duplicate chunks from sparse containers from a previous backup and record emerging sparse containers. They also manage container manifests that record which backups reference each container, and when a manifest becomes empty, a container can be safely removed. Such techniques could be used within our cloud GC algorithm, though copy-forward bandwidth is unlikely to be improved.

10 Conclusion

Data protection continues to be a key priority as customers transition their archival data to cloud storage. Data Domain is a mature data protection product, and adding a cloud tier involved trade-offs within the current architecture. We had to make decisions about object sizes and data structure relationships to balance performance and cost not only of migrating data to the cloud tier but also running GC. To address these concerns, we developed several techniques: mirroring metadata locally supports efficient deduplication and GC, and using perfect hashes to track billions of references in memory enables space estimation, seeding, and cloud GC. Experiences with initial customers shows a strong interest in deduplicated archival storage. Large amounts of data are transferred each month, which benefit from deduplication both in terms of faster transfer speeds but also lower storage costs.

Acknowledgments

We thank our shepherd Gala Yadgar and the anonymous reviewers. We thank the many Data Domain filesystem, QA and performance engineers who have contributed to its cloud tier-Bhimsen Bhanjois, Shuang Liang, Kalyani Sundaralingam, Jayasekhar Konduru, Kalyan Gunda, Ashwani Mujoo, Srikant Viswanathan, Chetan Rishud, George Mathew, Prajakta Ayachit, Srikanth Srinivasan, Lan Bai, Abdullah Reza, Kadir Ozdemir, Colin Johnson.

References

[1] ALLU, Y., DOUGLIS, F., KAMAT, M., PRABHAKAR, R., SHILANE, P., AND UGALE, R. Can't We All Get Along? Redesigning Protection Storage for Modern Workloads. In *USENIX Annual Technical Conference (ATC'18)* (2018).

[2] ALLU, Y., DOUGLIS, F., KAMAT, M., SHILANE, P., PATTERSON, H., AND ZHU, B. Backup to the future: How workload and hardware changes continually redefine Data Domain file systems. *Computer 50*, 7 (2017), 64–72.

[3] AMAZON. Amazon S3. https://aws.amazon.com/s3/, 2018. Retrieved Sept. 17, 2018.

[4] AMAZON. Amazon Web Services. https://aws.amazon.com/, 2018. Retrieved Sept. 17, 2018.

[5] BELAZZOUGUI, D., BOTELHO, F. C., AND DIETZFELBINGER, M. Hash, displace, and compress. In *Algorithms-ESA 2009*. Springer, 2009.

[6] BOTELHO, F. C., PAGH, R., AND ZIVIANI, N. Practical perfect hashing in nearly optimal space. *Information Systems* (June 2012). http://dx.doi.org/10.1016/j.is.2012.06.002.

[7] BOTELHO, F. C., SHILANE, P., GARG, N., AND HSU, W. Memory efficient sanitization of a deduplicated storage system. In *USENIX Conference on File and Storage Technologies (FAST'13)* (2013).

[8] DELL EMC. Elastic Cloud Storage. https://www.dellemc.com/en-us/storage/ecs/, 2018. Retrieved Sept. 17, 2018.

[9] DIETZFELBINGER, M., AND PAGH, R. Succinct data structures for retrieval and approximate membership. In *Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part I* (2008), ICALP '08, Springer-Verlag, pp. 385–396.

[10] DOUGLIS, F., BHARDWAJ, D., QIAN, H., AND SHILANE, P. Content-aware load balancing for distributed backup. In *Large Installation System Administration Conference (LISA)* (2011).

[11] DOUGLIS, F., DUGGAL, A., SHILANE, P., WONG, T., YAN, S., AND BOTELHO, F. C. The logic of physical garbage collection in deduplicating storage. In *USENIX Conference on File and Storage Technologies (FAST'17)* (2017).

[12] FORMAN, G., ESHGHI, K., AND SUERMONDT, J. Efficient detection of large-scale redundancy in enterprise file systems. *SIGOPS Oper. Syst. Rev. 43*, 1 (Jan. 2009), 84–91.

[13] FU, M., FENG, D., HUA, Y., HE, X., CHEN, Z., XIA, W., HUANG, F., AND LIU, Q. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *USENIX Annual Technical Conference (ATC'14)* (2014).

[14] HARNIK, D., HERSHCOVITCH, M., SHATSKY, Y., EPSTEIN, A., AND KAT, R. Sketching volume capacities in deduplicated storage. In *USENIX Conference on File and Storage Technologies (FAST'19)* (2019).

[15] JIN, K., AND MILLER, E. L. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of The Israeli Experimental Systems Conference* (2009), ACM.

[16] LI, J., LI, Y. K., CHEN, X., LEE, P. P., AND LOU, W. A hybrid cloud approach for secure authorized deduplication. *IEEE Transactions on Parallel and Distributed Systems 26*, 5 (2015), 1206–1216.

[17] LU, M., CHAMBLISS, D., GLIDER, J., AND CONSTANTINESCU, C. Insights for data reduction in primary storage: a practical analysis. In *Proceedings of the International Systems and Storage Conference* (2012), ACM.

[18] MAO, B., JIANG, H., WU, S., FU, Y., AND TIAN, L. Read-performance optimization for deduplication-based storage systems in the cloud. *ACM Transactions on Storage 10*, 2 (Mar. 2014).

[19] MEYER, D., AND BOLOSKY, W. A study of practical deduplication. In *USENIX Conference on File and Storage Technologies (FAST'11)* (2011).

[20] NAGESH, P., AND KATHPAL, A. Rangoli: Space management in deduplication environments. In *Proceedings of the International Systems and Storage Conference* (2013), ACM.

[21] NEELAVENI, P., AND VIJAYALAKSHMI, M. A survey on deduplication in cloud storage. *Asian Journal of Information Technology 13*, 6 (2014), 320–330.

[22] NG, W. K., WEN, Y., AND ZHU, H. Private data deduplication protocols in cloud storage. In *Proceedings of the ACM Symposium on Applied Computing* (2012), ACM, pp. 441–446.

[23] PAULO, J., AND PEREIRA, J. A survey and classification of storage deduplication systems. *ACM Computing Surveys 47*, 1 (2014).

[24] PUZIO, P., MOLVA, R., ONEN, M., AND LOUREIRO, S. Cloudedup: secure deduplication with encrypted data for cloud storage. In *International Conference on Cloud Computing Technology and Science (CloudCom)* (2013), vol. 1, IEEE, pp. 363–370.

[25] SRINIVASAN, K., BISSON, T., GOODSON, G. R., AND VORUGANTI, K. iDedup: latency-aware, inline data deduplication for primary storage. In *USENIX Conference on File and Storage Technologies (FAST'12)* (2012).

[26] SUN, Z. J., KUENNING, G., MANDAL, S., SHILANE, P., TARASOV, V., XIAO, N., AND ZADOK, E. Cluster and single-node analysis of long-term deduplication patterns. *ACM Trans. Storage 14*, 2 (May 2018), 13:1–13:27.

[27] TAN, Y., JIANG, H., FENG, D., TIAN, L., AND YAN, Z. Cabdedupe: A causality-based deduplication performance booster for cloud backup services. In *IEEE International Conference on Parallel & Distributed Processing Symposium (IPDPS)* (2011).

[28] VRABLE, M., SAVAGE, S., AND VOELKER, G. M. Cumulus: Filesystem backup to the cloud. *ACM Transactions on Storage (TOS) 5*, 4 (2009), 14.

[29] VRABLE, M., SAVAGE, S., AND VOELKER, G. M. Bluesky: A cloud-backed file system for the enterprise. In *USENIX conference on File and Storage Technologies (FAST'12)* (2012).

[30] WALLACE, G., DOUGLIS, F., QIAN, H., SHILANE, P., SMALDONE, S., CHAMNESS, M., AND HSU, W. Characteristics of backup workloads in production systems. In *USENIX Conference on File and Storage Technologies (FAST'12)* (2012).

[31] XIA, W., JIANG, H., FENG, D., DOUGLIS, F., SHILANE, P., HUA, Y., FU, M., ZHANG, Y., AND ZHOU, Y. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE 104*, 9 (Sept. 2016).

[32] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *USENIX Conference on File and Storage Technologies (FAST'08)* (2008).