Delta Compressed and Deduplicated Storage Using Stream-Informed Locality

Philip Shilane, Grant Wallace, Mark Huang, and Windsor Hsu
Backup Recovery Systems Division
EMC Corporation

Abstract

For backup storage, increasing compression allows users to protect more data without increasing their costs or storage footprint. Though removing duplicate regions (deduplication) and traditional compression have become widespread, further compression is attainable. We demonstrate how to efficiently add delta compression to deduplicated storage to compress similar (non-duplicate) regions. A challenge when adding delta compression is the large number of data regions to be indexed. We observed that stream-informed locality is effective for delta compression, so an index for delta compression is unnecessary, and we built the first storage system prototype to combine delta compression and deduplication with this technology. Beyond demonstrating extra compression benefits between 1.4-3.5X, we also investigate throughput and data integrity challenges that arise.

1 Introduction

Purpose-built backup storage has become widely deployed as a replacement for traditional tape backups. Deduplication in combination with local compression such as LZ has made hard drive systems cost competitive and enabled the transition from tape [11].

Further improvements to compression will enable users to protect more primary data within their backup storage while minimizing costs, data center space, and power. A promising technology is to apply delta compression, which compresses relative to similar (non-identical) regions of files. A key challenge is the large number of data regions to be indexed, which will not fit in memory for production systems. Previous work in delta storage systems assumed version information for files is known [3, 6] or a similarity index can fit in memory [1, 5]. Other work used an on-disk index [10] that requires random I/O for each query.

We observe that as backups change over time, a modified region is likely similar to the previous corresponding region and will be surrounded by unmodified regions that are duplicates. Based on these stream properties, we can load a cache of references, which eliminates the need for version information and large index structures. While our previous work investigated stream-informed locality for replication across the wide area network [9], that work was limited to low-throughput environments and did not attempt to store delta encoded chunks.

This project has four key contributions: 1) We present the first storage implementation of deduplication and delta compression using stream-informed locality. 2) We explore new complexities related to data integrity and cleaning. 3) We report the combination of deduplication and delta compression across a range of chunk sizes. 4) We quantify throughput and suggest areas for further improvement.

Stream-informed delta compression increases overall compression but also introduces new challenges that we begin to investigate. Delta compression is costly in the sense of requiring extra computation and I/O, but by applying deduplication first, delta compression on the remaining bytes becomes feasible. Besides throughput issues, delta compression introduces new architectural challenges because of the extra level of indirection. Protecting user backups is of utmost importance, so validating the delta encoded filesystem must be efficient. Also, a garbage collection algorithm must handle indirect references correctly. This paper presents these new complexities and initial results.

2 Delta Filesystem

2.1 Architecture

Deduplicating storage systems are used to remove redundant data thus improving storage efficiency [8]. In general a file is divided into content-defined chunks [7], which are fingerprinted with a strong hash. Each fingerprint is queried in an index to determine if the system already stores a copy of the chunk. If a copy exists then the current chunk need not be stored again and only a reference to it is recorded in the file recipe.

A primary challenge in deduplicating storage systems is managing a large index of fingerprints in such a way that queries are fast. These indexes can be 100's of GB in size thus requiring large portions to spill to disk and requiring effective caching techniques. The key insight of the Data Domain system [11] is that duplicate chunks appear in roughly the same stream-ordered patterns for backup workloads. In Figure 1, file backup.tar is initially written and divided into chunks 1-6. If minor changes occur to backup.tar then, the next time it is written, the chunks will appear in largely the same order. This stream-locality can be leveraged for deduplication purposes by grouping neighboring chunks together as a cache unit and loading the group's fingerprints whenever one of them is queried in the index.

A follow-on insight [9] is that non-duplicate chunks in a stream are often similar to chunks loaded by a neighboring chunk. Again referring to Figure 1, in the second version of backup.tar, chunk 4 has been slightly modified (i.e. chunk 4' in red) and will be very similar to chunk 4 in the cache. This similarity can be detected using resemblance hashes (hereafter sketches) which have the property that similar chunks will have identical sketches [2, 5, 10]. Briefly, a sketch is generated from maximal values within a chunk that tend to persist if a chunk is slightly modified. More details are presented in Shilane et al. [9]. If the new chunk is similar to an existing chunk, it can be delta encoded relative to the existing chunk. We call the encoded chunk a 'delta' and the existing referenced chunk its 'base'. The delta can then be stored on disk with a reference to its base chunk for decoding purposes, and this results in compression benefits.

Multi-level Delta

Delta encoding can span multiple levels of indirection. Chunk C could be encoded as 1-level delta of chunk B, which might in turn be a 1-level delta of chunk A. This causes C to be a 2-level delta, and arbitrary levels are possible.

As the level of delta encoding (n) varies from chunk to chunk, throughput will vary in unpredictable ways because n base chunks must be read for decoding [10]. Earlier work [9] investigated the compression differences between multi-level and 1-level delta encoding and reported an increase of 1.03 - 1.18X additional delta compression when the number of levels is not restricted. Based on these results, our current prototype implements 1-level delta, though future work could explore the tradeoff between compression and throughput further.

Prototype

We built a prototype storage system by modifying the deduplication architecture of the Data Domain system [11] to compute both fingerprints and sketches, which are stored together on-disk in caching units called containers. Thus whenever a container is loaded to the cache, both its fingerprints and sketches are loaded. When eviction occurs, based on an LRU policy, a container's fingerprints and sketches are evicted as a group. We used an 8KB average chunk size and 4.5MB containers holding chunks, fingerprints, and sketches.

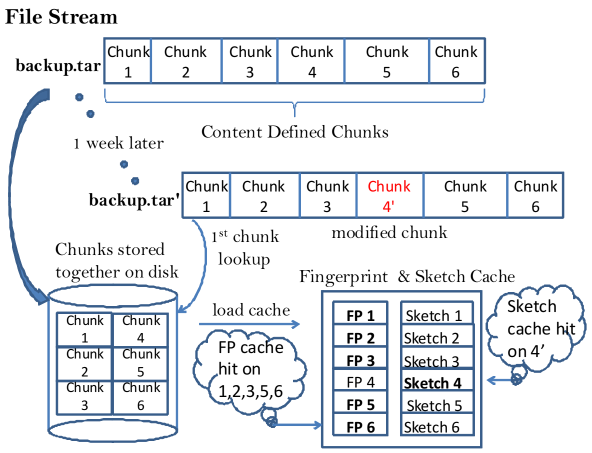

Figure 1: Modified chunks tend to be surrounded by identical chunks, which can be used to load a stream-informed cache with fingerprints and sketches from a caching unit called a container.

When a chunk is presented for storage, its fingerprint is compared against the cache and potentially an on-disk index for a match. If no fingerprint match is found, then we check for a similarity match by comparing the chunk's sketch against the cached sketches. If a match is found, then we read in the base chunk, delta encode the current chunk, compress the delta, and store the result. By introducing delta compression to the pipeline we are able to achieve higher total compression than deduplication and local compression can achieve. Our stream-informed delta technique does not require an additional sketch index, but it may miss potential similarity matches when cache locality degrades.

2.2 Compression Results

Our experiments include four datasets described in Table 1 that are 2-5 TB in size and span 4-6 months of collected backups. These datasets consist of large tar-type files that consist of many user files concatenated together by backup software in a pattern of repeated full and incremental backups. `Source Code` is backups of a version control repository for source code. `Workstations` is backups from 16 software engineers' desktops. `Email` is daily full backups from a MS Exchange server. `System Logs` is backups from a server's /var directory.

Table 1 reports compression factors as contributed by deduplication, delta, LZ, total compression, and delta improvement. Metadata overhead of 30 bytes per chunk is included in these results. Compression factors are calculated as $input\_bytes/output\_bytes$ for each compression stage, so values greater than 1 indicate a compression improvement. Total compression is a multiplication of the three previous columns since they are independent factors. A subtle point is that delta and LZ compres-

Dataset	Size TB	Months	Dedup.	Delta	LZ	Total	Delta Impr.
Workstations	2.3	4	5.0	4.2	1.6	33.6	3.5
Email	2.5	5	4.9	2.6	2.1	26.8	2.1
Source Code	4.5	6	16.7	3.6	2.5	150.3	1.4
System Logs	5.3	4	25.2	3.3	1.8	149.7	1.5

Table 1: Compression factors for backup datasets.

sion overlap because delta encoding removes redundancies within a chunk itself [9], so the listed LZ compression value is lower than when delta is disabled. The delta improvement column factors in the change in LZ compression to show the true improvement of adding delta compression is between 1.4X to 3.5X beyond a baseline of deduplication with LZ.

These delta compression values are the result of our stream-informed sketch cache, which has been shown to produce compression comparable to a full sketch index without the additional data structures [9].

3 Practical Considerations

While delta compression can add significant storage savings, there are numerous practical issues that must be resolved to build a production system. We investigate throughput, garbage collection, the impact of varying chunk size, and data integrity.

3.1 Throughput

Delta compression adds extra compression at the cost of extra computation and I/O as discussed in earlier work [9, 10]. There is extra computation to create sketches, lookup sketches in a cache, and perform delta encoding. As CPU cores continue to increase, computation may become less of a concern.

There is also extra I/O required to read in a chunk from disk to serve as a base, so write throughput is limited by the underlying I/O performance of the system. Chunks are typically stored compressed so a read must also decompress the data, and if larger regions are read together, good locality can amortize the I/O time.

Results

We performed an experiment writing the four datasets both with and without delta compression enabled. During the first full backup, the throughput was 74% (std. deviation 14%) of the deduplicated storage system with the decrease likely related to sketching and some extra disk I/O and computation. After the first full backup, delta compression increases because there is sufficient previous data to compress against, and throughput decreased to 53% (std. deviation 18%) of the deduplicated storage baseline.

We further investigated throughput bottlenecks in Table 2, which shows single-stream timing for each stage of the delta algorithm. We should emphasize that this is an unoptimized prototype running on a system with Intel

Dataset	Sketch MB/s	Lookup MB/s	Encode MB/s	Read alternatives HDD MB/s	Read alternatives SSD MB/s
Workst.	47	1,528	94	5	400
Email	49	1,441	69	1	80
Src. Code	30	30	31	2	160
Sys. Logs	30	70	50	2	160

Table 2: Timing by delta stage including alternatives for reading base chunks.

dual quad-core Xeon 2.6Ghz processors. Our research suggests that optimization will likely improve the lookup and read stages as discussed below.

Looking for matches in the sketch cache takes place at 30 MB/s - 1.5 GB/s, which is highly variable. We determined that `Source Code` and `System Logs` had very long chains of identical sketch matches in the cache. A 50X difference in speed is attributable to traversing 50X more links in a hashtable, so limiting duplicate sketches in the cache would likely improve this stage.

Next, reading back a base chunk from disk is clearly the bottleneck in processing because the prototype did synchronous single-stream reads. Throughputs as low as 1 MB/s correspond to $\sim$10 ms disk seeks for each 8KB chunk synchronously, and higher throughputs are related to caching data. One solution is to move from hard drives to SSD type storage with faster I/O characteristics. We derived throughput results for the SSD case assuming the same memory caching properties from the actual read experiment and an access latency of 100 us. For 8KB chunks, with no caching, this corresponds to a throughput of about 80 MB/s. However, there is an order of magnitude increase in cost associated with moving to SSD storage.

Aggregate throughput is higher than these per-stage numbers indicate for multiple reasons. Importantly, deduplication takes place before delta encoding and effectively applies a multiplier to the throughput. Also, only about $2/3$ of post-deduplication chunks have a similarity match and require base reads and encoding. Finally, multi-threading and asynchronous reads across multiple disks would give significant benefit.

3.2 Garbage Collection

To have a complete backup storage system, removing deleted files and the underlying chunks is a necessary feature unlike in some archival systems [10]. When implementing a cleaning algorithm, the first concern is always data integrity. Because of deduplication and delta compression, there are numerous references to stored chunks, and we implemented a background deletion process called garbage collection (GC).

Our filesystem is log-structured, so cleaning involves copying live chunks forward. There are two standard techniques for tracking live chunks: reference counts and mark-and-sweep. Reference counts are updated as fingerprint and delta base references are added and removed from the system, but reference counts have known resiliency issues in a dynamic system that may experience complicated failure cases. The other alternative is to record live chunks in the GC mark stage and copy those chunks forward during the sweep stage.

Duplicates and Similar Chunks

A deduplicated storage system might choose to write occasional duplicates because of memory constraints or to improve performance [4]. In combination with delta compression, complicated data patterns may result in chains of delta encodings.

For example, suppose chunk *A* is stored and chunk *B* is stored as a delta of *A* since it is similar. At a later time, chunks *B* and *A* are written again. While they could be identified as duplicates of their earlier versions, suppose instead that *B* is written normally (a duplicate). When *A* is written the second time, it could be written as a delta of the base version of *B*.

This leads to multiple possible paths for reading chunks *A* and *B*. One path involves reading the version of *A* that is delta of *B*, which requires reading *B*, which could be the version that is in turn a delta of the standard version of *A*. These cases become complicated, and improper handling can result in reference loops and data loss.

Results

An issue specific to stream-informed delta compression is that as chunks are copied forward, we are effectively changing data locality and combining chunks from different regions into containers. Since we do not have a sketch index, we may miss out on potential delta compression because of degraded locality.

We measured the amount of incremental delta compression each week without GC as our baseline. We then reran the test using a four week retention policy, ran GC to free deleted chunks, and measured delta compression each week.

Figure 2 shows how much delta compression was achieved when GC ran relative to the baseline without GC. For `Email` and `Source Code`, delta compression was nearly 100% of the no-GC case with a standard deviation near zero. `Workstations` achieved 80% of the baseline compression with a standard deviation of 18.

Relocating chunks during GC can affect locality for loading the sketch cache, but we are also changing the amount of total data available as base chunks. In future work, we would like to further investigate the relationship between these two concerns These results indicate some impact of GC, but the delta compression achieved is still a significant benefit.

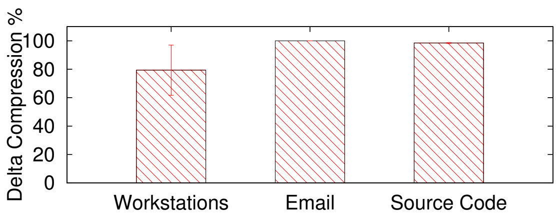

Figure 2: Garbage collection has a modest impact on delta compression.

3.3 Compression vs. Chunk Size

Increasing the chunk size has the potential to improve throughput because of larger read/write units, and it may ease memory pressures related to tracking references per chunk, but there is a risk of losing compression with larger chunks.

For this experiment, we designed a simulator that can more easily handle a configurable chunk size than our prototype. Figure 3 shows compression for deduplication, delta, LZ, and total compression across chunks sizes from 1KB to 1MB (x-axis). Logscale is used so that small values for delta and LZ are observable. Note that we used 8-10 full backups for these experiments as compared to the full datasets in earlier experiments. Metadata overhead was included and has the largest impact on small chunks.

For comparison, we show the amount of total compression achieved without delta. Note that LZ contributes somewhat higher compression in this case because delta does not remove similar regions as discussed in Section 2.2.

Deduplication is highest with small chunk sizes and decreases steadily as the chunk size increases. Delta compression starts off slightly above 1 (no extra compression) and grows steadily as the chunk size increases because it finds compression that deduplication is now missing. Total compression reaches its peak at approximately 8KB average chunk size but is generally flat in that region largely due to delta compression offsetting decreased deduplication.

These results indicate that delta compression adds significantly increased compression across chunk sizes. Also, we could increase the chunk size to 64KB while maintaining compression similar to our typical 8KB system. We leave performance evaluations across chunk size as future work.

3.4 Data Integrity

Our delta filesystem is targeted for backup storage, and the integrity of user data is the highest priority. Besides RAID at the storage layer, it is necessary to add end-to-end checks that files can be read back correctly. This means confirming that all chunks referenced by file recipes exist and that every chunk is valid.

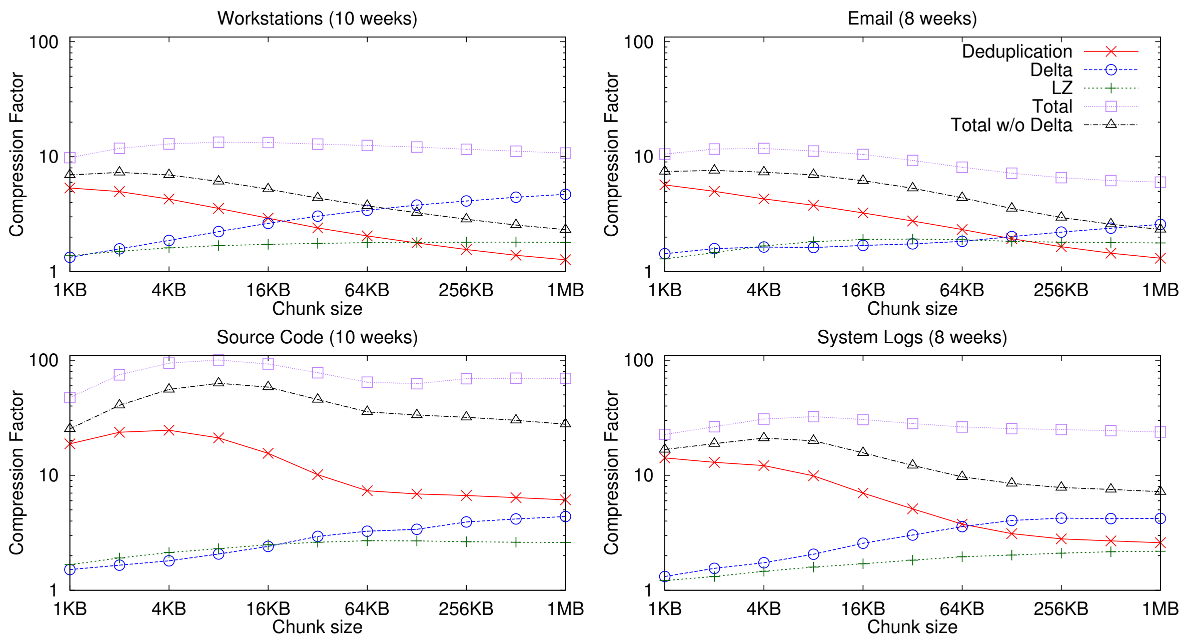

Figure 3: As chunk size increases, deduplication decreases and delta compression increases to compensate.

It is important to validate that there is no corruption either in the delta encoding or in the decoded chunk. Confirming the validity of the delta encoding can be handled with a checksum. However, validating the decoded chunk adds a level of complexity, which requires a read for the base chunk adding significant overhead. We have begun to address data integrity issues but leave a detailed discussion for future work.

4 Conclusion and Future Work

In this paper, we present a delta compressed and deduplicated storage prototype, which demonstrates that stream-informed locality enables both efficient deduplication and delta compression. Delta compression adds an additional 1.4X-3.5X compression beyond deduplication and local compression and would allow users to protect more data within a storage system.

We also explore challenges introduced by adding delta compression to the storage system such as throughput bottlenecks, cleaning, and ensuring data validity. Our analysis shows large I/O overheads when reading base chunks, which could be addressed by replacing hard drives with SSDs. There are new complexities when cleaning deleted chunks, and we present results indicating that the stream-informed cache necessary for delta compression is mildly affected. We also demonstrate that delta compression adds significant compression across chunk size, though the impact on throughput is left as future work.

This prototype is an effective demonstration, and future research could improve and simplify techniques for increasing throughput, cleaning, validating data.

References

[1] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein. The design of a similarity based deduplication system. In *SYSTOR*, 2009.

[2] A. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences*, 1997.

[3] R. C. Burns and D. D. E. Long. Efficient distributed backup with delta compression. In *Workshop on I/O in parallel and distributed systems*, 1997.

[4] F. Guo and P. Efstathopoulos. Building a high-performance deduplication system. In *USENIX Annual Technical Conference*, 2011.

[5] P. Kulkarni, F. Douglis, J. LaVoie, and J. M. Tracey. Redundancy elimination within large collections of files. In *USENIX Annual Technical Conference*, 2004.

[6] J. MacDonald. File system support for delta compression. Master's thesis, Dept. of Electrical Engineering and Computer Science, Univ. of California at Berkeley, 2000.

[7] U. Manber. Finding similar files in a large file system. In *USENIX Winter Technical Conference*, 1994.

[8] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Conference on File and Storage Technologies*, 2002.

[9] P. Shilane, M. Huang, G. Wallace, and W. Hsu. WAN optimized replication of backup datasets using stream-informed delta compression. In *Conference on File and Storage Technologies*, 2012.

[10] L. You, K. Pollack, D. D. E. Long, and K. Gopinath. Presidio: A framework for efficient archival data storage. *ACM Transactions on Storage*, 7(2), July 2011.

[11] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *Conference on File and Storage Technologies*, 2008.