Reverse Deduplication: A Novel Deduplication Optimized for Restoration

Zhike Zhang, Ignacio Corderi, Darrell D.E. Long, Preeti Gupta
Storage Systems Research Center
University of California at Santa Cruz
1156 High Street, Santa Cruz, CA 95064

Abstract

Data deduplication has become an important part of the data storage industry, with most major companies providing products in the space. As additional data is added to a deduplicated storage system, the number of shared data chunks increases. This leads to the randomization of the data in the system, which in turn leads to increased seek operations and decreased performance.

The challenge for all companies is to provide high performance both at the time of data ingest, and when the data is retrieved. In many cases, the primary use of deduplicating storage systems is to provide an alternative to tape-based back-up. For these systems, performance during ingest is important, and the most common retrieval case is the most recent back-up. But due to the nature of existing deduplication algorithms, the most recent back-up is also the most fragmented, resulting in performance issues. Also in traditional deduplication, the chunks are reference counted. A system that requires following so many references will result in poor performance when restoring newer versions.

We address this issue by changing the way deduplication is done. We propose Reverse Deduplication. We also suggest algorithms to eliminate much of the fragmentation in the most recent backups and we introduce the notion of a chunk descriptor. A chunk descriptor would be associated with a chunk signature, and would be reference counted. It would have a pointer to the actual physical chunk. In the case when the physical location of the chunk moves, only this descriptor would need to be updated. A key advantage of using chunk descriptors is that we do not have to traverse a bunch of pointers.

By adding this level of indirection, we avoid following chains of reference Our results show that, the average ratio of the retrieval time taken by Reverse Deduplication to that taken by the traditional deduplication is 0.69.

1 Introduction

Data deduplication is an important part of the storage industry today helping backup systems survive with less storage space. But performance is of key importance to systems using deduplication both at the time of ingest as well as during data retrieval. Most common retrieval case is last backup retrieval. However traditional deduplication approaches make the last backup the most fragmented. We suggest a change in the deduplication methodology.

Currently, as new data segments are added, they are deduplicated against the existing corpus and duplicate chunks are not stored. The result is that newer segments have greater opportunity to find existing chunks, and so may be increasingly fragmented as the storage system grows. Instead, we propose to invert the deduplication process. Each new segment will be written contiguously, and older data segments sharing chunks in the new segment will reference those chunks. As a result restoring the most recent copy, will be the most efficient. We are doing post-process or asynchronous deduplication, done after the backups have been stored on the disks.

Efficiently managing the chunks is also essential for performance. Typically, the chunks are reference counted where chunks might be referenced by several generations of a data segment. A system that requires following so many references will result in poor performance when restoring older versions.

One possibility would be to restructure the data as part of the cleaning process, but another possibility which is likely to be quicker is to introduce the notion of a chunk descriptor. Consider Figure 1, A chunk descriptor would be associated with a chunk signature, and would be reference counted. It would have a pointer to the actual physical chunk. In the case when the physical location of the chunk moves, only this descriptor would need to be updated. By adding this level of indirection, we avoid following chains of references.

Experimental results show that, the average ratio of the retrieval time taken by Reverse Deduplication to that taken

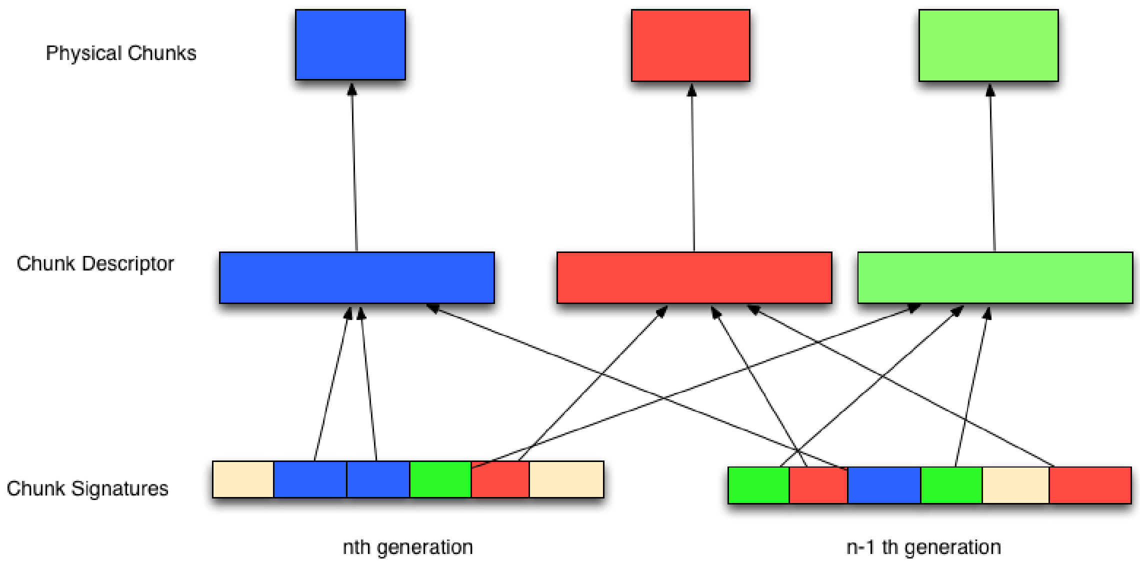

Figure 1: shows chunk signatures pointing to chunk descriptors which are pointing to physical chunks.

by the traditional deduplication is 0.69, while Reverse Deduplication needs almost twice storage utilization than the traditional deduplication.

In this paper we demonstrate the existence of fragmentation in deduplicated systems and quantify its extent and effect on performance and develop algorithms that will provide the best performance for the common use cases,in particular restoring the most recent back-up from the archive.

2 Related Work

The simplest form of deduplication is whole file hashing. This technique computes a cryptographic hash function (such as MD5 or SHA-1) for every file in the storage system [1, 2]. Only files with a unique cryptographic hash are stored, with duplicate files replaced by a pointer to existing files. The chief benefit of whole file hashing is speed; deduplication requires only minimal metadata, but it suffers from the limitation that even a single changed bit requires the entire file to be stored.

To address this limitation of whole file hashing [1], techniques have been developed that deduplicate files at a finer granularity. These techniques deterministically break files into chunks, which can either be fixed [3] or variable length [4, 2, 5], and store only the unique chunks of the file. As with whole file hashing, a hash function (typically a cryptographic hash function) is computed for each chunk of the file. This hash value, called the chunk signature, uniquely identifies the data chunk.

Fixed-size chunking suffers from a common limitation related to insertions: a single byte inserted causes all subsequent chunk hash values to change. To address this issue, variable-length chunking can be employed. Variable length chunking uses a deterministic algorithm to establish each chunk boundary by calculating these boundaries based on the content of a file. Variable-length chunking can result in chunks that are either too small, or too large. To address this issue, algorithms such as Two Threshold Two Denominators (TTTD) have been developed [6].

All deduplication techniques suffer from an increase in metadata information, the magnitude of which depends on the deduplication technique. Since the index of chunks quickly grows so large that it is impossible to keep it in memory, the goal then becomes to minimize the number of disk operations required to access the index. Recent research has shown that it is possible to greatly reduce the number of disk operations, while only minimally impacting the degree of deduplication [7, 8, 5]. To decrease the amount of metadata, the chunking algorithm namely bi-modal chunking [9] uses small chunks in transition regions from duplicate data to new data, and uses large chunks elsewhere. Big chunks and also their subchunks are both deduplicated [10].

The performance of single node deduplication is limited. Some deduplication clusters are presented for scalable throughput and scalable capacity. A distributed deduplication system [7] is designed based on Extreme Binning. HYDRAstor [11] is a deduplication cluster and uses a distributed hash table to routes chunks to storage nodes. HydraFS [12] is a file system based on HYDRAstor. The super-chunk is proposed and an routing algorithm is designed based on the super-chunk [13].

As the data in the system increases, so does the fragmentation of the data. The degree to which this occurs is believed to be worse than in a traditional file system. Some have proposed performing periodic defragmentation. Disk defragmentation is more difficult in the presence of deduplication, since before relocating a data block, the file system must make sure that all references to the block are updated as well. Macko, et al. [14] propose the use of back references to solve problems such as finding all file i-nodes in a deduplicated system that reference a block being defragmented. These back references can be used to prevent the file system from having to iterate over the set of all files to find all referenced blocks. Other techniques, such as reference counting, can also be employed.

3 Analysing Fragmentation

To maximize the writing throughput, most dedup systems sequentially store new chunks on the disk, such as Bloom Filter [5], Sparse Indexing [8] and Extreme Binning [7]. A chunk store contains a number of chunk containers. The chunk container contains data chunks, and some metadata including the offset of chunks and ownership of chunks and so on. When one chunk container is full, a new chunk container is created. The size of chunk container is 4 GB typically. Traditional deduplication techniques cause fragmentation since new duplicate chunks are removed.

We used File Fragment Degree (FFD) to quantize the fragmentation of a file in a deduplication system. FFD is the number of areas one file scatters on the disk. e.g. if a file is stored contiguously on the disk, FFD is 1 but if it is stored in three non-contiguous areas in disks, FFD is 3. Higher FFD means more seeks to get the file. In this section, we try to answer these questions:

- What is the average FFD of the deduplication system?
- What is the distribution of the FFD of the deduplication system?
- What is the trend of the average FFD as the deduplication system grows?

For our experiments we used three datasets. Each dataset represents a different type of workload. The first dataset is the linux source code archive, namely *Linux*. It contains versions from 1.2.0 to 2.5.75, totally 564 versions. Most files of *Linux* are small files, typically dozens of KB. *Linux* represents high redundant dataset consisting of small files. The second dataset *HDup* is having 162 full backups and 416 incremental backups collected from 21 engineers in 30 days. *HDup* represents high redundant dataset consisting of large files. To simulate an incremental only backup, we extracted the first full backup and all the incremental backups for every engineer from *HDup* to construct our third dataset *LDup*. *LDup* consists of 21 full backups and 416 incremental backups, representing low redundant dataset. The first full backup represents data when users backup their PCs for the first time. Rest of the workload represents backup requests for later changes. We call this set LDup since it contains few duplicates HDup and LDup are chunked data however Linux data is being chunked using Two Threshold Two Denominators (TTTD) algorithm [16]. TTTD performs better than the basic sliding window chunking algorithm in finding duplicate data [17]. The average size of the chunks was 4 KB and the chunks were not compressed.

Table 1 summarizes the size information of the three datasets.

Table 1: Data Sets Used in Experiments

Set	Files	Size	Unique Files
Linux	3186361	35.678 GB	184164
HDup	17669935	4540.209 GB	1034353
LDup	2273779	946.359 GB	970461

Table 2: Summary of FFD for Various Datasets

Dataset	Min.	Average	Max.	95% Perc.	99% Perc.
linux	1	1.706	75	5	9
hdup	1	4.406	18108	2	11
ldup	1	3.601	16750	2	9

3.1 Statistics of FFD

Table 2 shows statistics of FFD for the datasets we used.

The average FFD for *HDup* and *LDup* is large. Note that, these two datasets represent backup data for only one month. Common deduplication systems are designed to store backup data of several years. We believe that, for a deduplication system reduplicating data for several years, the average FFD will be quite large. This would deteriorate the reading or restoring performance.

The average FFDs of *Linux* is not large, because it mainly contains small files. But it can still affect the reading performance.

The average FFDs of *HDup* and *LDup* are much higher than that of *Linux*. Because *Linux* mainly consists of small files, typically from several kilobytes to dozens of kilobytes, and *HDup* and *LDup* contains large files. Most files in *Linux* may only contains several data chunks.

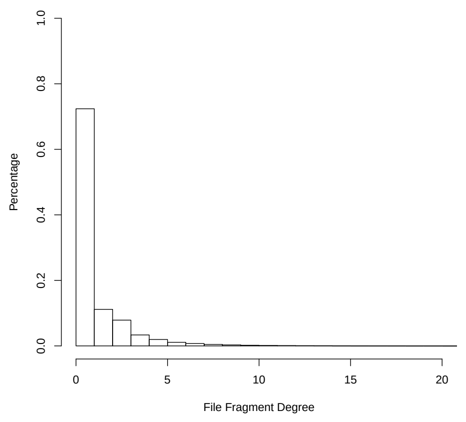

Figure 2: Histgram of FFD for Linux. The y axis represents the percentage of the files with a specific FFD. For example, the first bar on the left shows what the percentage of the files with FFD 1 in all files is.

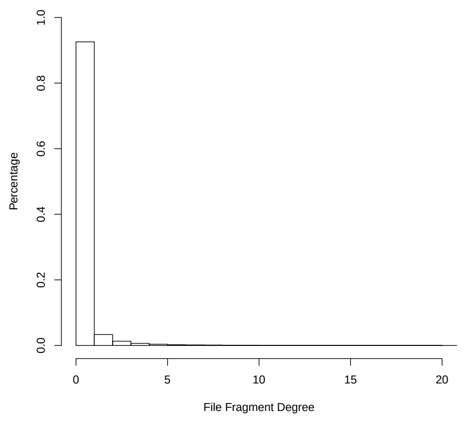

Figure 3: Histgram of FFD for HDup. The y axis represents the percentage of the files with a specific FFD. For example, the first bar on the left shows what the percentage of the files with FFD 1 in all files is.

3.2 FFD Distributions for Various Datasets

Figure 2, Figure 3 and Figure 4 present the FFD distributions for *Linux*, *HDup* and *LDup*. Very few files have FFD more than 5. The percentage of FFD decreases as FFD value increases.

For *Linux*, there are about 30% files with FFD more than 2. For *HDup* and *LDup*, there are about 10% files with FFD more than 2.

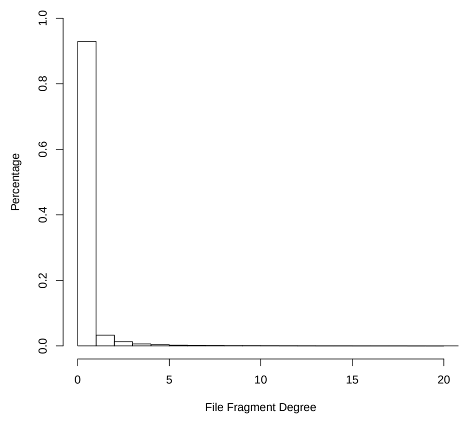

Figure 4: Histgram of FFD for LDup. The y axis represents the percentage of the files with a specific FFD. For example, the first bar on the left shows what the percentage of the files with FFD 1 in all files is.

3.3 Trends of the Average FFD for Various Datasets

Figure 5, Figure 6 and Figure 7 show the trends of the average FFD for various datasets.

The average FFDs for *HDup* and *LDup* increase rapidly as the deduplication system grows larger. This indicates that, as the dedeplication grows larger and larger, the reading or restoring performance can be significantly deteriorated probably by the fragments. Initially, average FFD goes down, because backup data containing many new files is being stored in the system. As the system matures, there are not so many new files any more, and the average FFD grows smoothly.

The average FFDs of *HDup* and *LDup* increase much faster than that of *Linux*. Because *Linux* mainly contains small files. We believe that, the data fragments of deduplication system for backup data mainly containing small files does not significantly deteriorate the reading or restoring performance.

It shows that data fragmentation is present in the deduplication system for various datasets. For enterprise backup data, the amount of data fragments increases rapidly as the deduplication system grows however for dataset mainly containing small files, it increases slowly as the deduplication system grows. More data fragments mean worse retrieval performance. Therefore, we need to reduce the fragmentation of deduplication systems to improve the performance of retrieval operations.

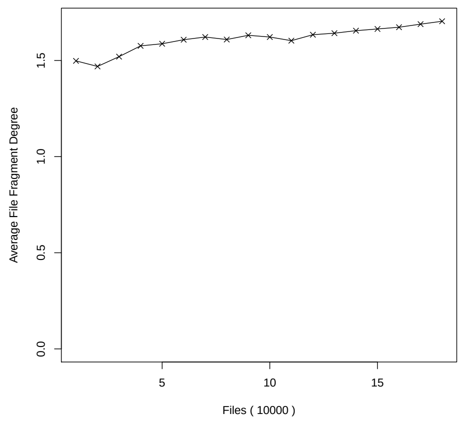

Figure 5: Trend of the average FFD for Linux, as files are ingested by the deduplication system. We can see that, as more files are deduplicated and stored, the average FFD increases slowly. It means the fragmentation increases slowly.

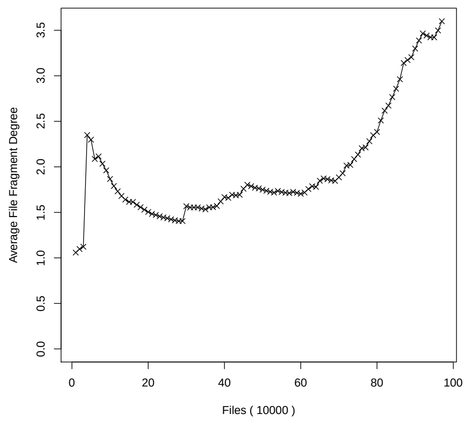

Figure 7: Trend of the average FFD for LDup, as files are ingested by the deduplication system. We can see that, as more files are deduplicated and stored, the average FFD increases rapidly especially after about 60,000 file are deduplicated. It means the fragmentation increases rapidly.

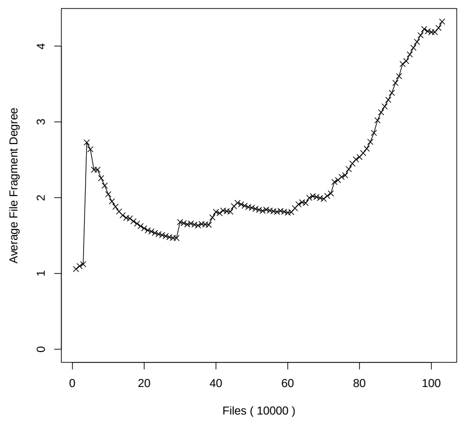

Figure 6: Trend of the average FFD for HDup, as files are ingested by the deduplication system. We can see that, as more files are deduplicated and stored, the average FFD increases rapidly especially after about 60,000 file are deduplicated. It means the fragmentation increases rapidly.

4 Design

Deduplication saves storage space by sharing duplicate data chunks. This results in the randomization of data in disks leading to more disk seeks and worse performance when retrieving data. In the existing deduplication systems [5, 8, 7], new data segments are deduplicated against the existing data segments, and duplicate data chunks are not stored. The newer data segments has greater chance to find duplicate data chunks. This leads to the most recent back-up, referencing the data chunks in the older backups, the most fragmented and the most slow.

To solve this performance issue, we propose to invert the deduplication process. All data chunks of every new data segment will be written to disks contiguously, older data segments share and reference data chunks in the newest data segments. Our algorithm can eliminate much of the fragmentation of the most recent backup.

Figure 8 shows how Reverse Deduplication works. The byte stream contains three backups. Every backup contains three data chunks. When the second backup segment is stored, traditional deduplication finds chunk 2 and chunk 3 are duplicate, thus only writes chunk 4 to disk. But Reverse Deduplication writes chunk 2, chunk 3 and chunk 4 to disk, and reclaims the old chunk 2 and the old chunk 3. The same thing happens when the third backup, containing chunk 1, chunk 4 and chunk 5, is stored. Each of the two algorithms finds chunk 1 and chunk 4 are duplicate. The traditional deduplication writes only the new chunk 5, but Reverse Deduplication writes all three chunks and also reclaims the old chunk 1 and the old chunk 4. We can clearly see that, the most recent backup is completely fragmented in the traditional deduplication and completely contiguous in Reverse Deduplication. For quick reference, we summarize the naive Reverse Dedu-

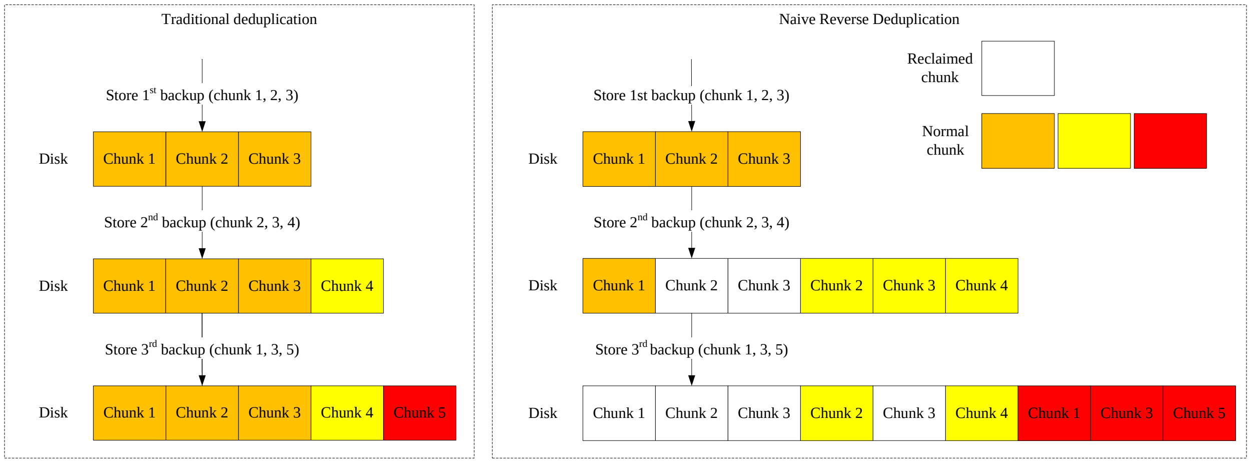
Traditional deduplication
Store 1st backup (chunk 1, 2, 3)
Disk
Chunk 1 Chunk 2 Chunk 3
Store 2nd backup (chunk 2, 3, 4)
Disk
Chunk 1 Chunk 2 Chunk 3 Chunk 4
Store 3rd backup (chunk 1, 3, 5)
Disk
Chunk 1 Chunk 2 Chunk 3 Chunk 4 Chunk 5
Naive Reverse Deduplication
Reclaimed chunk
Normal chunk
Store 1st backup (chunk 1, 2, 3)
Disk
Chunk 1 Chunk 2 Chunk 3
Store 2nd backup (chunk 2, 3, 4)
Disk
Chunk 1 Chunk 2 Chunk 3 Chunk 2 Chunk 3 Chunk 4
Store 3rd backup (chunk 1, 3, 5)
Disk
Chunk 1 Chunk 2 Chunk 3 Chunk 2 Chunk 3 Chunk 4 Chunk 1 Chunk 3 Chunk 5

Figure 8: Disk layout of data chunks of the naive Reverse Deduplication and the traditional deduplication. We can clearly see that, the most recent backup, which consists the data chunk 1 and 3 and 5, is completely fragmented in the traditional deduplication and completely contiguous in Reverse Deduplication.

plication in Algorithm 1.

Algorithm 1 Naive Reverse Deduplication Algorithm

```
Input: byte stream
  chunk the byte stream;
  for each chunk do
    writeChunkToDisks(chunk);
    dupChunk = findDupChunk(chunk);
    if dupChunk != NULL then
      reclaimPriorDupChunk(dupChunk);
    end if
    updateChunkIndex();
  end for
```

When naive Reverse Deduplication is used, the most recent backup is still probably fragmented. Since if there are duplicates within the same backup, this algorithm will deduplicate them as well causing the fragmentation. Also it deduplicates the duplication between most recent backups of all the users. We are trying to make most recent backup contiguous on the disk. So we modified naive deduplication algorithm not to claim the duplicate data amongst most recent backups of different users and also within itself. This leads to more storage utilization, but it can ensure all the most recent backups are contiguous in disks. For quick reference, we summarize the optimized reverse deduplication in Algorithm 2.

5 Experimental Results

We compared Reverse Deduplication and the traditional deduplication by the restoration time and also the deduplication efficiency. We used HDup, as shown in subsection 3 to evaluate Reverse Deduplication. The most recent backup of every engineer is a full backup, except one engineer.

Algorithm 2 Reverse Deduplication Algorithm

```
Input: byte stream
  chunk the byte stream;
  for each chunk do
    writeChunkToDisks(chunk);
    dupChunk = findDupChunk(chunk);
    if (dupChunk != NULL) && dupChunk is not in this
    user's most recent backup && dupChunk is not in
    every other user's most recent backup then
      reclaimPriorDupChunk(dupChunk);
    end if
    updateChunkIndex();
  end for
```

5.1 Simulator

We implemented simulators for Reverse Deduplication and traditional deduplication. The traditional deduplication deduplicates new data segments against old data segments and duplicate data chunks are not written to disks. Reverse Deduplication writes all data chunks of every new data segment to disks contiguously, older data segments share and reference data chunks in the newest data segments. Both the two algorithms read the data from disk, deduplicate the data, and then store necessary meta-data including data chunk addresses. They both maintain complete index in RAM. The signature of each data chunk is deduplicated against the complete index.

We retrieve the last backup of each user, using the addresses of data chunks. We used 16GB cache. 8 MB data is pre-fetched. The size of data block is 4 KB. One data chunk may span two data blocks In this case, we read all data blocks spanned by the data chunk. We use LRU as the cache replacement algorithm. The number of disk seeks and the size of data read from disks are recorded. We use these numbers to compute the time taken by the retrieval operation.

Cryptographic hash or chunk ID of the chunk is computed using techniques such as MD5 [18] or SHA [19], [20].

5.2 Evaluation Metrics

We used the restoration time of the last backup to compare the retrieval performance between Reverse Deduplication and traditional deduplication. Reverse Deduplication is designed to provide good retrieval performance for the last backup. The restoration time is computed by Equation 1. Denote the restoration time by $t_{retrieve}$, the disk seek number by n_{seek}, the disk seek time by t_{seek}, the disk rotation latency by $t_{latency}$, the size of data read from disks by s_{read}, the sustained transfer rate of disk by r_{disk}.

$$t_{retrieve} = n_{seek} \times (t_{seek} + t_{latency}) + s_{read} \div r_{disk} \quad (1)$$

We used a desktop class disk, the Barracuda from Seagate [15] in our simulation and compute the amount of disk time in seconds, where t_{seek} is 9 ms, $t_{latency}$ is 4 ms, and r_{disk} is 156 MB/s. Our simulator outputs n_{seek} and s_{read} for retrieval operations.

We used the compression rate to evaluate the storage utilization, as shown in Equation 2. The $deduplicated\_size$ is the storage utilization. The $original\_size$ is the size of data stored in the system.

$$compression\_rate = \frac{deduplicated\_size}{original\_size} \quad (2)$$

5.3 Comparing Restoration Time of Last Backups

Our algorithm is designed to benefit the last backup. We compared the restoration time of the last backup for each user between Reverse Deduplication and traditional deduplication. The dataset is firstly deduplicated. Then, we retrieve the last backup for each user of the 21 users in the dataset. Our simulator records the number of disk seeks and the size of data read from disks when a backup is retrieved. We use these numbers to compute the restoration time by Equation 1.

The results are shown in Table 3. 9. The average ratio of the time taken by Reverse Deduplication to the time taken by the traditional deduplication is 0.69. Reverse Deduplication performs much better than the traditional deduplication when retrieving the last backups because it has contiguous data on disks.

Table 4: Comparison of compression rates between Reverse Deduplication and the traditional dedup

Alg.	Orig. (GB)	Stor. (GB)	Compr. Rate
Rev.	4540.21	584.81	0.13
Trad.	4540.21	299.35	0.07

Table 5: The most recent full backup of every engineer is deduplicated separately using the traditional deduplication. The average rate of duplicate data is 0.41.

User ID	Orig.(GB)	Dup.(GB)	Dup. Rate
0	16.77	4.68	0.28
1	7.23	2.22	0.31
2	43.24	35.57	0.82
3	14.16	5.05	0.36
4	9.22	3.70	0.40
5	26.12	5.01	0.19
6	106.27	24.04	0.23
7	8.98	3.65	0.41
8	55.73	16.76	0.30
9	28.96	14.52	0.50
10	52.57	13.02	0.25
11	7.40	2.13	0.29
12	8.18	6.16	0.75
13	3.64	0.66	0.18
14	8.13	5.98	0.74
15	35.08	15.59	0.44
16	34.40	10.78	0.31
17	27.02	5.58	0.21
18	22.00	10.52	0.48
19	34.12	23.45	0.69
20	13.70	5.63	0.41

5.4 Comparison of Deduplication Rate

Table 4 shows the comparison of compression rates between Reverse Deduplication and the traditional dedup. Reverse Deduplication needs about twice storage than the traditional deduplication. It is worse than the traditional method in this aspect.

5.5 The size of duplicate data in most recent backup for each user

Table 5 shows the size of duplicate data in most recent full backup for each user. Each of them is deduplicated using the naive Reverse Deduplication. The average rate

Table 3: Comparing restoration time of last backups. The total time taken by Reverse Deduplication to restore the last backups for each user is 4629.66 seconds. The total time taken by traditional deduplication is 6671.05 seconds. The last column shows the ratio of the time taken by Reverse Deduplication to the time take by the traditional deduplication. The average ratio is 0.69.

	Reve. Dedup			Trad. Dedup			
User ID	Seek Nr.	Read Size (GB)	Time (s)	Seek Nr.	Read Size (GB)	Time (s)	Reve./Trad.
0	2147.00	12.13	138.01	4644.00	17.30	193.86	0.71
1	929.00	5.25	59.72	2959.00	11.22	125.08	0.48
2	5545.00	31.31	356.23	16182.00	61.69	686.45	0.52
3	1813.00	10.25	116.54	3184.00	13.62	146.55	0.80
4	1181.00	6.67	75.92	2153.00	10.38	108.07	0.70
5	3504.00	19.54	222.85	7426.00	33.71	356.70	0.62
6	13532.00	76.47	869.86	16368.00	91.63	919.97	0.95
7	1150.00	6.50	73.92	3359.00	14.04	152.00	0.49
8	7169.00	40.42	460.03	13151.00	58.01	618.70	0.74
9	3711.00	20.97	238.55	6193.00	29.51	308.26	0.77
10	6730.00	38.03	432.62	14646.00	63.91	683.66	0.63
11	948.00	5.36	60.94	2004.00	10.12	104.17	0.59
12	896.00	5.06	57.60	1261.00	5.57	59.39	0.97
13	467.00	2.64	30.02	3548.00	14.35	156.85	0.19
14	1042.00	5.89	66.98	1356.00	6.11	64.79	1.03
15	4500.00	25.40	288.99	7349.00	31.97	342.30	0.84
16	4446.00	25.07	285.30	8098.00	39.14	407.39	0.70
17	3461.00	19.56	222.48	8047.00	36.57	386.83	0.58
18	2823.00	15.95	181.42	5171.00	22.78	243.03	0.75
19	4339.00	24.52	278.92	8234.00	39.72	413.57	0.67
20	1754.00	9.91	112.75	4559.00	17.39	193.45	0.58

of duplicate data is 0.41.

6 Conclusion

We demonstrated that fragmentation is existing in all the three data sets we used though it is much higher for large full data backups in comparison with incremental data backups and Linux backups and grows exponentially as data increases. With reverse deduplication approach optimized for last backup makes it completely contiguous be it for any user. Our results show that the average ratio of the retrieval time taken by Reverse Deduplication to that taken by the traditional deduplication is 0.69, while Reverse Deduplication needs almost twice storage utilization than the traditional deduplication.

7 Future Work

Due to the reverse chunking scheme that we propose, older data segments will develop holes (portions of the data segment that are no longer referenced). The data accesses will be less contiguous, and so performance for older segments will decrease. As a result, it will become necessary to garbage collect these segments in order to reclaim space and increase the contiguity of the segments. The reclamation process can happen in an off-line fashion, similar to the segment cleaner in a log structured file system. In fact, the changes to these older segments can also be done in a lazy fashion by writing change records to a log. The older segment remains valid as is, and applying the change record serves to release storage space associated with the older segment. We will also address the question of how to manage the holes that will develop in older segments. The naive approach would be to deduce them from the segment or file descriptor (like an i-node), but that is certainly not an efficient approach. A better approach might be, for example, to have a bit map that shows internal (part of the contiguous segment) or external (belonging to another segment) chunks. Bits in this map would be set when deduplication happens. As a result, only internal chunks need to be copied when rewriting the segment.

8 Acknowledgements

References

[1] N. Jain, M. Dahlin, and R. Tewari, "Taper: Tiered approach for eliminating redundancy in replica synchronization," in *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies*. San Francisco, CA, USA: USENIX Association, Berkeley, CA, USA, 1416 December 2005, pp. 281–294.

[2] L. You, K. Pollack, and D. Long, "Deep store: An archival storage system architecture," in *Proceedings of the 21th International Conference on Data Engineering*. Tokyo, Japan: IEEE Computer Society, Washington, DC, USA, 5-8 April 2005, pp. 804–815.

[3] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage," in *Proceedings of the Conference on File and Storage Technologies*. Montery, CA, USA: USENIX Association, Berkeley, CA, USA, 28-30 January 2002, pp. 89–101.

[4] S. Rhea, R. Cox, and A. Pesterev, "Fast, inexpensive content-addressed storage in foundation," in *Proceedings of the 2008 USENIX Annual Technical Conference*. San Diego, CA, USA: USENIX Association, Berkeley, CA, USA, 14-19 June 2008, pp. 143–156.

[5] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proceedings of the 6th Conference on USENIX Conference on File and Storage Technologies*. San Jose, CA, USA: USENIX Association, Berkeley, CA, USA, 26-29 February 2008, pp. 269–282.

[6] G. Forman, K. Eshghi, and S. Chiocchetti, "Finding similar files in large document repositories," in *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Chicago, IL, USA: ACM, New York, NY, USA, 21-24 August 2005, pp. 394–400.

[7] D. Bhagwat, K. Eshghi, D. Long, and M. Lillibridge, "Extreme binning: Scalable, parallel deduplication for chunk-based file backup," in *Proceedings of the 17th Annual Meeting of the IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*. London, UK: IEEE Computer Society, Washington, DC, USA, 21-23 September 2009, pp. 1–9.

[8] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble, "Sparse indexing: Large scale, inline deduplication using sampling and locality," in *Proceedings of the 7th Conference on USENIX Conference on File and Storage Technologies*. San Francisco, CA, USA: USENIX Association, Berkeley, CA, USA, 24-27 February 2009, pp. 111–123.

[9] E. Kruus, C. Ungureanu, and C. Dubnicki, "Bimodal content defined chunking for backup streams," in *Proceedings of the 8th Conference on USENIX Conference on File and Storage Technologies*. San Jose, CA, USA: USENIX Association, Berkeley, CA, USA, 23-26 February 2010, pp. 239–252.

[10] B. Romański, Ł. Heldt, W. Kilian, K. Lichota, and C. Dubnicki, "Anchor-driven subchunk deduplication," in *Proceedings of the 4th Annual International Conference on Systems and Storage*. ACM, 2011, p. 16.

[11] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki, "Hydrastor: A scalable secondary storage," in *Proceedings of the 7th Conference on USENIX Conference on File and Storage Technologies*. San Francisco, CA, USA: USENIX Association, Berkeley, CA, USA, 24-27 February 2009, pp. 197–210.

[12] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Całkowski, C. Dubnicki, and A. Bohra, "Hydrafs: A high-throughput file system for the hydrastor content-addressable storage system," in *Proceedings of the 8th Conference on USENIX Conference on File and Storage Technologies*. San Jose, CA, USA: USENIX Association, Berkeley, CA, USA, 23-26 February 2010, pp. 225–238.

[13] W. Dong, F. Douglis, K. Li, H. Patterson, S. Reddy, and P. Shilane, "Tradeoffs in scalable data routing for deduplication clusters," in *Proceedings of the 9th Conference on USENIX Conference on File and Storage Technologies*. San Jose, CA, USA: USENIX Association, Berkeley, CA, USA, 15-17 February 2011, pp. 15–29.

[14] P. Macko, M. Seltzer, and K. Smith, "Tracking back references in a write-anywhere file system," in *Proceedings of the 8th USENIX conference on File and storage technologies*. USENIX Association, 2010, pp. 2–2.

[15] S. Inc., "Barracuda sata product manual," December 2012. [Online]. Available: http://www.seagate.com

[16] G. Forman, K. Eshghi, and S. Chiocchetti,"Finding similar files in large document repositories, in *KDD 05: Proceeding of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, 2005, pp. 394400.

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

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

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

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

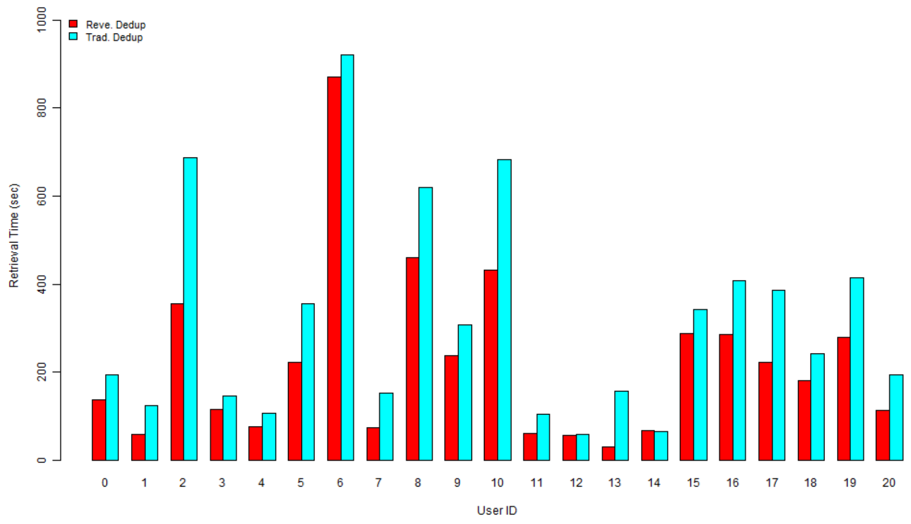
Reve. Dedup
Trad. Dedup
Retrieval Time (sec)
User ID
0
200
400
600
800
1000
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Figure 9: