Rethinking Deduplication Scalability

Petros Efstathopoulos
Petros_Efstathopoulos@symantec.com

Fanglu Guo
Fanglu_Guo@symantec.com

Symantec Research Labs
Symantec Corporation, Culver City, CA, USA

1 ABSTRACT

Deduplication, a form of compression aiming to eliminate duplicates in data, has become an important feature of most commercial and research backup systems. Since the advent of deduplication, most research efforts have focused on maximizing deduplication efficiency—i.e., the offered compression ratio—and have achieved near-optimal usage of raw storage. However, the capacity goals of next-generation Petabyte systems requires a highly scalable design, able to overcome the current scalability limitations of deduplication. We advocate a shift towards scalability-centric design principles for deduplication systems, and present some of the mechanisms used in our prototype, aiming at high scalability, good deduplication efficiency, and high throughput.

2 INTRODUCTION

Enterprise storage is rapidly shifting to disk-based backup, loading deduplication systems with the burden of dealing with the uncontrolled explosion of data that corporations are required to keep. Research [1, 2] and commercial systems, such as Symantec PureDisk [3], have utilized a number of different variations of techniques and deduplication principles: file segmentation methods, index optimizations, various forms of caching, and many other mechanisms have been heavily optimized in order to maximize duplicate detection and deduplication efficiency.

Despite achieving very high deduplication efficiency, commercial systems today still suffer from limited scalability, and have difficulties scaling to Petabyte-level capacities. Once a system reaches its scalability limits, administrators can do very little to increase system capacity and avoid significant performance degradation. Scalability plans based on adding more nodes, introduce serious system management and performance problems. We believe that an optimized deduplication engine is insufficient, if the system cannot scale to high capacities. Therefore, we are attempting to design a next-generation deduplication system, from the ground up, setting scalability as one of our top goals—even at the cost of less than optimal deduplication efficiency.

2.1 Hitting the Memory Wall

The core of any deduplication system, performs two basic tasks: it *detects* and *shares* duplicate data blocks—instead of storing multiple copies. When operating in small scale, we can easily maintain a block index in memory to quickly check if a block already exists in the system, and use methods similar to object reference sharing within operating systems to keep track of data sharing. Challenges arise when the deduplication system needs to scale to high capacities—and billions of objects.

Item	**Scale**	**Remarks**
Physical capacity C	C = 400 TB	
Segment size B	B = 4 KB	
Number of segments N	N = 100 Gsegs	$N = C/B$
Block fingerprint size F	F = 20 B	
Block index size I	I = 2000 GB	$I = N * F$
Disk speed Z	Z = 300 MB/sec	
Block lookup speed goal	75 Kops/sec	Z/B

Table 1: An example system configuration, illustrating some of the challenges involved.

When a deduplication server is presented with a piece of data it needs to answer the following questions: "Have I seen this data before? And if I have, where is it stored?" This kind of lookup operation is served using an indexing data structure–often referred to as "*the index*". A hash is calculated for each unique chunk of data stored in the system (MD5, SHA-1/2, etc), and stored in the index as the chunks *fingerprint*, along with the chunk's location on disk.

During a backup on a typical block-level deduplication system, all files are partitioned to segments. Segment size can vary between a few hundred bytes to multiple KB. For each segment, a fingerprint FP is calculated, and if FP does not exist in the index, a copy of the data is stored in the system, and the index is updated. This process might have to be performed for millions of segments during a backup.

Table 1 presents an example which gives a sense of the target scale and the challenges of a large scale system. In this example, the performance requirements dictate that we need to maintain the segment fingerprint index in main memory, but its size is simply too large (2000 GB) to fit[1]. Storing the index on disk is not a viable solution, because disk random access speed can not support 75 Kops/sec. Additionally,

[1]Notice that the 20 bytes do not include additional block metadata (location, flags, etc). For a 20-byte fingerprint we would require a total of at least 25 bytes per block entry.

segment fingerprints are cryptographic hashes, randomly distributed in the index, and adjacent index entries share no locality among them. Therefore, segment read-ahead and simple caching cannot make up for low disk performance.

2.2 Resource Reclamation

Contrary to traditional backup systems, where each file consists of its own data blocks, a deduplication system shares data blocks by default. Whenever a file is deleted, we need to determine whether each of its data blocks is still in use. If not, we can reclaim the segment and reuse the space.

The simplest method to solve this problem is reference counting for segments: a segment's reference count is incremented every time it is used by a file, and decremented when the data segment is released—eventually reclaiming the segment when the count drops to zero.

Reference counting is less suitable for deduplication for several reasons. First, if we want to build a multi-node large scale deduplication system, reference counting will need to be transactional—in order to support references to remote data segments—leading to serious performance degradation. Second, even in a single-node system, it is not trivial to make a simple reference count work correctly: any lost or repeated update will incorrectly change the count. Logging is necessary to recover from these error conditions. If a data object becomes corrupted, however, it is desirable to know which files are using it, so as to request new copies of those files, and recover the corrupted data. Unfortunately, reference counting cannot help us determine which files are using a particular data segment.

Maintaining a reference list is a better solution, since it is immune to repeated reference updates, since a reference list can determine whether the reference add/remove operation in question has been performed already. Furthermore, reference lists have the capability to identify which files are using each data segment. However, although reference lists are immune to repeated operations, they cannot deal with lost operations and transactions, and some kind of logging is still necessary to ensure correctness. Additionally, maintaining a variable-length reference list has heavy space and computational overhead, since we need to persistently store it on disk. Also, managing individual reference lists for each of the billions of data segments, is simply too costly, while managing a single reference list for all data segments would require to allocate space for new entries on every addition, or rewrite the whole list. In either case, there is no simple solution.

Another alternative is the mark-and-sweep approach. During the mark phase, we need to go through all files and mark all data segments that are in use. Then, in the sweep phase, we sweep the data segment lists, reclaiming all unmarked segments. The main advantage of mark-and-sweep is that it is very resilient to errors. If anything goes wrong, we can simply restart the process, and all operations can be repeated without side effects. The main downside, however, is that it is not scalable: going back to the example of Table 1, we would need to deal with N = 100 Gsegments. If a file uses fingerprints (F = 20 bytes) to reference data segments, we are going to read at least $N * F$ = 2,000 GB of data in order to mark all files once, assuming each data segment is used once. In a typical system, where each data segment will be used an average of 10 times (deduplication factor), marking once would require reading 20,000 GB of data. Assuming disk speed of 300 MB/sec, marking alone will take 18.5 hours. Since mark-and-sweep needs to touch all files in the system, the larger the system, the slower the process becomes.

3 TOWARD A SCALABLE DESIGN

When considering a scalable deduplication server design, we have to take into consideration the following intertwined metrics: speed, scale, and space. In order to reduce space usage, we do extra work for data segment lookup and sharing. This impacts backup speed. Which, in turn, can be especially difficult to achieve when the system reaches a larger scale. Keeping these three metrics in mind, we define the following as our goals:

- Scalability: support hundreds of billions of segments. Ideally, we would like to support an unlimited number of data segments.
- Capacity: perform best effort deduplication. If resources are scarce, we are willing to sacrifice some space for speed and scale.
- Speed: near raw disk throughput for backup, restore, and data removal.

Even though space savings is deduplication's primary purpose, it is not the only goal. Speed (i.e., high throughput) is also important, because a system that is too slow to finish a backup within a backup window, is useless. Scalability is important, since a highly scalable system can greatly decrease the management cost, by reducing the number of computers involved. Thus speed and scalability are more a usability issue. We aim at making our system usable, and then try our best to save space under the resource constraints.

3.1 Indexing: Beyond Memory Bounds

Notice that successfully locating a fingerprint in the index during backup only affects the deduplication efficiency of the server. For example, if a lookup fails, even though the segment in question is already stored in the system, a duplicate copy of the segment will be stored, but correctness will be preserved. Using this kind of flexibility, we apply sampling techniques and maintain only a subset of fingerprints in the memory index.

For our sampled index we assume a sampling period T, signifying that we insert in the index only "1 out T" new fingerprints. We define a sampling rate R as follows:

$$R = 1/T = (S * M)/(E * C) \tag{1}$$

where M is the amount of memory available for indexing (in GB), S is the deduplication segment size (in KB), E is the memory entry size (in bytes), and *C* is the total supported storage (in TB). For instance, in the example of Table 1, using a generous 32 bytes per index entry (20 bytes for the fingerprint + 12 bytes of entry metadata), with 4 KB segments and 32 GB of memory available for indexing, we can support 4 TB of data with a sampling rate of 1 (i.e., no entries are dropped). Scaling to 400 TB, would require a sampling rate of 0.01—i.e., insert in the index one out of 100 fingerprints. Using an 8 KB segment size, the sampling rate doubles to 0.02 (one out of 50 segments), sacrificing some index accuracy (deduplication efficiency) for higher scalability. This sampling scheme allows us to scale to a theoretically "infinite" capacity: expanding the system's storage capacity without upgrading its indexing capacity (i.e., amount of RAM), comes at the cost of lower sampling rates (i.e., lower deduplication efficiency). Investing in indexing capacity (by adding more RAM), is rewarded with higher sampling rates.

3.1.1 *Spatial Locality and Pre-fetching*

Using sampled indexing we are able to scale to higher storage capacity, but the index hit-rate—and, consequently, the deduplication efficiency—would be *T* times lower. To address this problem, we are relying on the spatial locality among data segments: during backup, adjacent segments are stored in the same disk container—even if some of the relevant fingerprints are not sampled for insertion in the index. Upon an index hit, we locate the container pointed by the index entry, and temporarily pre-fetch that container's internal fingerprint index into memory. The likelihood of subsequent lookups hitting on these temporary index entries is very high, due to spatial locality properties of data streams—previously observed and taken advantage of in many deduplication systems (e.g., [4]). Using part of main memory to implement this type of container index caching is significantly improves the deduplication efficiency, even when relatively low sampling rates are utilized.

3.1.2 *Progressive Sampling*

Sampled indexing and container index pre-fetching provide us with a scalable, well performing indexing solution with reasonable deduplication efficiency. We can further improve the deduplication efficiency, by observing that formula 1 calculates the minimum rate required to support the total amount of available storage. Alternatively, we can calculate a *progressive sampling rate*, by using the amount of storage currently *used*—as opposed to the total capacity. For instance, in the previous example, if the system is equipped with 400 TB of storage, but only 500 GB are being used, there is no reason to utilize the rate R = 0.01, since we can easily index all 500 GB of data stored in the system. As the amount of used storage increases, we can adjust the sampling rate and easily re-sample the index, by dropping the extra entries. For instance, we can start with R = 1 (all FPs are inserted in the index), and once the amount of utilized storage approaches 8 TB, we can switch to R = 0.5, by dropping half the entries in the index (e.g., use "mod T" sampling).

3.1.3 *Solid State Drives for Solid Scalable Indexing*

Using sampled indexing, we are able to scale to higher capacities, but main memory size remains a limiting factor for scalability. Additionally, we need to maintain a disk copy of the memory index, in order to provide persistence and crash recovery. Maintaining and querying the index directly on hard drives is not considered a viable alternative, for performance reasons, but Solid State Drives (SSDs) provide an alternative that may prove a valuable indexing tool.

SSDs support high storage capacities, with unique performance characteristics: SATA SSDs are able to achieve sequential read/write throughput of around 250/170 MB/sec, and about 35,000/3,300 IOPS for random 4 KB reads/writes [5]. High-end, PCIe SSDs [6] can achieve read/write performance of 750/500 MB/sec and around 120,000/90,000 random IOPS, respectively. Our prototype aims to leverage SSDs and provide an alternative to memory indexing: the fingerprint index is stored on the SSD, and it is possible to locate a fingerprint with at most one SSD read operation. In order to amortize the cost of low SSD write performance (dominated by long flash memory erase cycles), index updates are logged, buffered and batched. Additionally, container indexes are pre-fetched and cached, in order to amortize some of the SSD read cost. By storing the primary index on the SSD, we achieve the following benefits:

- Better deduplication: sampling rate calculated using SSD capacity.
- Memory benefits: large amounts of RAM no longer needed—available RAM used primarily for caching.
- Full indexing: a large SSD can support a full index
- Index persistence: index persistently stored at all times

Notice that when using an SSD index, the memory would be used primarily for caching purposes, as well as for storing a Bloom filter large enough to summarize the SSD.

3.2 Grouped Mark-and-Sweep

In order to make mark-and-sweep a scalable solution, we must reduce the number of files we need to touch during the process. Based on the observation that the majority of files in the system are not changed between backups—and therefore we do not need to include them in the mark phase over and over again—we propose the *grouped mark-and-sweep* algorithm, presented in Figure 1.

Each backup, consists of a list of files, and one or more backups form a group. We track changes to each group of backups, and for each changed group we further track if files are added to or deleted from it. In Figure 1, we assume that some files were deleted from Group1 and some files were added to Group3, while Group2 remained unchanged since the last mark-and-sweep cycle. G1, G2, and G3 are the mark results of each group's containers, and can be thought of as bitmaps showing which data blocks in the container are used by the files in a particular group. For example, G1 over SO container 1 shows which data blocks in container 1 are used

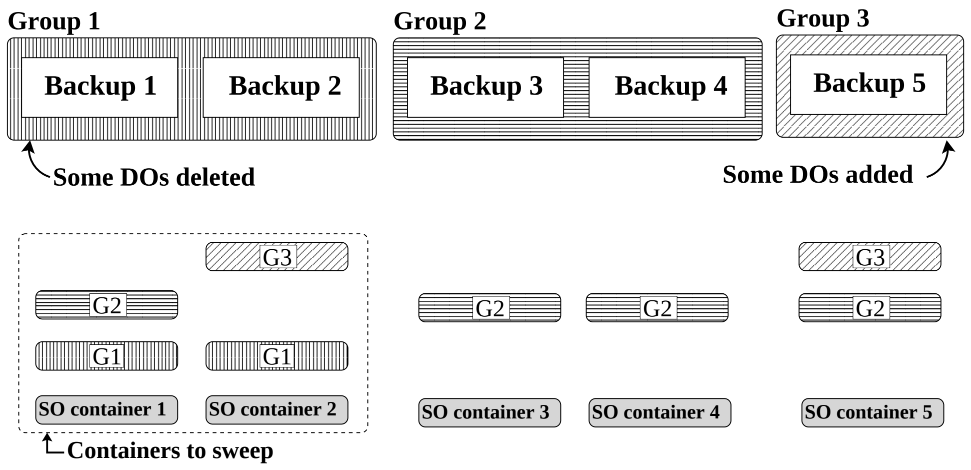

Figure 1: Example illustrating the scalability of our grouped mark-and-sweep mechanism.

by files from Group1. In our prototype, one mark-and-sweep cycle would involve the following steps:

- Go through all the changed groups and mark. Since Group1 and Group3 have been changed, we clear their old mark results, i.e., G1 and G3, for all containers. Go through all the files in Group1 and Group3. Generate new G1 and G3 for all containers that have data blocks used by files in Group1 and Group3, respectively.
- During the mark phase, track all containers which are marked from groups (Group1) that have files deleted. Put the containers to the sweep list. Only containers in the sweep list may have data blocks freed because only these containers are used by groups that have deleted files.
- After all files in changed groups (Group1 and Group3) are marked, sweep containers in the sweep list. In our example, it will be SO container 1 and SO container 2. For each container in the sweep list, merge the mark results of all groups for that container. If a data block is not used, it can be reclaimed.

Generalizing the above example, our approach takes the following steps to make mark-and-sweep scalable:

- Divide files to groups. Track changes to each group. Only mark the changed groups and avoid repeatedly marking the unchanged groups. Store the mark results. For unchanged groups, the old mark results can be reused.
- Track the affected containers. Only containers storing data blocks used from groups that have deleted files are put to sweep list because only these containers may have data blocks freed. During sweep, mark results are merged to determine if a data block is still in use.

In the above process, the workload is proportional only to the working set, as opposed to being proportional to the capacity of the whole system, since we avoid touching unchanged data, while still achieving the benefits of mark-and-sweep.

4 PRELIMINARY EVALUATION

We have implemented a full prototype of our scalable deduplication system design for Linux. Although our prototype implementation is still in progress, we are already able to perform full backups and evaluate the effectiveness of our scalability improvements.

4.1 Indexing

The current index implementation uses a split hash table design: one hash table (“the index”) is used to hold all sampled entries, and a separate hash table (“the cache”) is used to cache pre-fetched entries. Hash table buckets can be one or a few KB in size, and different strategies can be used for collision resolution: we want to avoid dropping colliding entries in the index, therefore we use chaining for hash table buckets—incurring a small performance penalty. However, the cache does not have such strict requirements, but we aim at high performance. When a particular bucket is full, we clear the bucket and make room for new entries. We keep track of already pre-fetched container indexes, in order to avoid unnecessary pre-fetching.

A lot of effort was invested in optimizing memory usage: index entry size has been reduced to sizes between 18 or 19 bytes, and all pointers have been substituted with offsets, leading to memory savings, and making the index easily serializable on disk.

Table 2 shows that index performance depends a lot on the index state: when using chaining, index operations become slower as the index fills up. However, even at high index load, index performance is well within our performance goals: on a 3 GHz Intel Xeon machine, we are able to achieve throughput of 229, 394 and 178 Kops/sec for lookup, insert and delete operations respectively—much faster than our goal of 75 Kops/sec. Assuming a 4 KB segment, this performance corresponds to 1,576, 916 and 712 MB/sec, just for sampled index entries. However, this high performance comes at a small cost: a small percentage of entries may need to be dropped, if the index runs out of buckets for colliding en-

X	Load	Insert	Lookup	Remove
1 M	0.7%	4,825	4,783	8,443
10 M	7%	5,411	5,320	8,923
100 M	70%	10,412	6,701	13,807
145 M	97%	13,101	7,620	16,836

Table 2: Average insert/lookup/remove cost, in CPU cycles, when the index is pre-populated with *X* entries. Index load calculated based on total capacity of 149 Mentries.

	SATA SSD	ioDrive
Seek time	0.24 msec	0.06 msec
Throughput	90 MB/sec	454 MB/sec
Cycles per lookup	138,871	55,590
Ops per sec	17,323	53,646

Table 3: Unbuffered SSD performance. Ops/sec assume a 3 GHz CPU.

tries. Performance presented in Table 2 is using a configuration guaranteed to suffer less than 2.3% of dropped entries.

The SSD index implementation was tested on a low-end SATA SSD, as well as a 160 GB SLC PCIe ioDrive. Table 3 presents the results of our preliminary SSD tests for full fingerprint lookup, demonstrating that our approach can be a viable, well-performing solution—especially after we introduce container caching.

4.2 Throughput and Dedup Performance

We have tested our prototype on a Linux server, using a 8 TB disk array with a raw read/write throughput of 305/330 MB/sec. Backup throughput was initially CPU-bound, due to the overwhelmingly expensive hash calculations. After implementing multi-threaded hash calculation, the backup server is able to achieve 98% of raw-disk backup throughput of for new backups (326 MB/sec). Deduplication backup tests, with container pre-fetching enabled, and using sampling rates of 10% and 14%, yielded perfect deduplication for consecutive backups of identical files—even though only a small subset of file fingerprints were sampled. With perfect deduplication, we observed backup throughput of up to 663 MB/sec. At very high capacity and index loads (above 90%), deduplicated backup throughput drops to a performance near the disk read throughput. The exact reasons are being investigated, but we believe it is due to disk read limitations (for container index pre-fetching) and a high number of cache flushes. Recent, ongoing testing on better hardware (more CPU cores, faster Fiber Channel arrays) showed that our design is limited by I/O performance—and not by inherent throughput limitations of our design.

4.3 Resource Reclamation

Table 4 shows time and throughput measurements for our grouped mark-and-sweep. After adding or removing data, our grouped mark-and-sweep thread runs to update the mark results for new and/or deleted backups. If any files or backups are deleted, it further checks whether any containers

Data	Add Time (Throughput)	Delete Time (Throughput)
30 GB	4.29 sec (6.99 GB/sec)	21.77 sec (1.38 GB/sec)
300 GB	39.33 sec (7.63 GB/sec)	218.77 sec (1.37 GB/sec)
990 GB	163.02 sec (6.07 GB/sec)	690.29 sec (1.43 GB/sec)

Table 4: Grouped mark-and-sweep resource reclamation is scalable: processing time is proportional to the size of the working set and throughput is stable regardless of working set size and system capacity.

can be reclaimed. We performed three different backups (30 GB, 300 GB, and 990 GB), measuring grouped mark-and-sweep performance after each backup. Then we removed each backup one by one, measured the mark-and-sweep processing time for each removal, and found it to be proportional to the amount of data added or deleted. The throughput is consistent—regardless of the size of the data added or removed. We repeated these tests with our test system running near its capacity (8 TB, 2 billion data segments), and got the same results, demonstrating that our resource reclamation method is indeed scalable.

5 CONCLUSIONS AND FUTURE WORK

Our work explores methods to overcome the scalability limitations of deduplication servers. We have addressed two very important scalability problems: sampled indexing can be used to overcome memory limitations, while grouped mark-and-sweep can significantly improve resource reclamation.

Our preliminary results have been encouraging, demonstrating that our techniques can indeed help us achieve high scalability, and very good throughput. We aim to investigate these and other methods further, pursuing maximum scalability for our prototype. In particular, among other things, our future work will focus on improving index performance, investigating the effectiveness of a full SSD index implementation, and introduce multi-threaded implementations of all CPU-intensive operations that can be parallelized.

REFERENCES

[1] Sean Quinlan and Sean Dorward, "Venti: A new approach to archival storage," in *FAST '02: Proceedings of the Conference on File and Storage Technologies*, Berkeley, CA, USA, 2002, pp. 89–101, USENIX Association.

[2] OpenDedup, "A userspace deduplication file system (SDFS)," Mar. 2010, `http://code.google.com/p/opendedup/`.

[3] Symantec, "Symantec NetBackup PureDisk," `http://www.symantec.com/business/netbackup-puredisk`.

[4] Benjamin Zhu, Kai Li, and R. Hugo Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *FAST '08: Proceedings of the Conference on File and Storage Technologies*, 2008, pp. 269–282.

[5] Intel Corporation, "Intel X25-E Extreme SATA Solid-State Drive," May 2009, `http://download.intel.com/design/flash/nand/extreme/319984.pdf`.

[6] Fusion-IO Corporation, "ioDrive," 2010, `http://www.fusionio.com/products/iodrive/`.