Building a High-performance Deduplication System

Fanglu Guo　　*Petros Efstathopoulos*
Symantec Research Labs
Symantec Corporation, Culver City, CA, USA

Abstract

Modern deduplication has become quite effective at eliminating duplicates in data, thus multiplying the effective capacity of disk-based backup systems, and enabling them as realistic tape replacements. Despite these improvements, single-node raw capacity is still mostly limited to tens or a few hundreds of terabytes, forcing users to resort to complex and costly multi-node systems, which usually only allow them to scale to single-digit petabytes. As the opportunities for deduplication efficiency optimizations become scarce, we are challenged with the task of designing deduplication systems that will effectively address the capacity, throughput, management and energy requirements of the petascale age.

In this paper we present our high-performance deduplication prototype, designed from the ground up to optimize overall single-node performance, by making the best possible use of a node's resources, and achieve three important goals: *scale* to large capacity, provide good *deduplication efficiency*, and near-raw-disk *throughput*. Instead of trying to improve duplicate detection algorithms, we focus on system design aspects and introduce novel mechanisms—that we combine with careful implementations of known system engineering techniques. In particular, we improve single-node scalability by introducing *progressive sampled indexing* and *grouped mark-and-sweep*, and also optimize throughput by utilizing an event-driven, multi-threaded client-server interaction model. Our prototype implementation is able to scale to billions of stored objects, with high throughput, and very little or no degradation of deduplication efficiency.

1 Introduction

For many years, tape-based backup solutions have dominated the backup landscape. Most of their users have been eager to replace them with disk-based solutions that are faster, easier to use (search, restore, etc.) and less fragile. In the past few years, disk-based backup systems have gained significant momentum, and today most enterprises are rapidly adopting such solutions, especially when the data volume is moderate.

One of the most important factors enabling the recent success of disk-based backup is data *deduplication* ("dedupe")—a form of compression that detects and eliminates duplicates in data, therefore storing only a single copy of each data unit. By using dedupe in a disk-based backup system one can multiply the effective capacity by 10-50 times, rendering the system a realistic tape replacement, whose cost is on par with tape-based systems, while also 1) making backup data always available online (for indexing, data mining, etc.), 2) enabling effective remote backups by minimizing network traffic, and 3) reducing client side I/O overhead by eliminating the need to read unchanged, previously backed-up files.

The explosive increase in the amount of data corporations are required to store, however, puts great pressure on the storage and backup systems, creating immediate demand for new ways to address the capacity, performance and cost challenges, and generally increase their overall effectiveness.

The effectiveness of a deduplication system is determined by the extent to which it can achieve three mutually competing goals: *deduplication efficiency*, *scalability*, and *throughput*. Deduplication efficiency refers to how well the system can detect and share duplicate data units—which is its primary compression goal. Scalability refers to the ability to support large amounts of raw storage with consistent performance. Throughput refers to the rate at which data can be transferred in and out of the system, and constitutes the main performance metric.

All three metrics are important. Good dedupe efficiency reduces the storage cost. Good scalability reduces the overall cost by reducing the total number of nodes since each node can handle more data. High throughput is particularly important because it can enable fast backups, minimizing the length of a backup window. Among the three goals, it is easy to optimize any two of them, but not all. To get good deduplication efficiency, it is necessary to perform data indexing for duplicate detection. The indexing metadata size grows linearly with the capacity of the system. Keeping this metadata in memory, would yield good throughput. But the amount of available RAM would set a hard limit to the scalability of the system. Moving indexing metadata to disk would remove the scalability limit, but significantly hurt performance. Finally, we can optimize for both throughput and scalability, as in regular file servers, but then we lose deduplication. Achieving all three goals is a non-trivial task.

Another less obvious but equally important problem is duplicate reference management: duplicate data sharing introduces the need to determine who is using a particular data unit, and when it can be reclaimed. The computational and space complexity of these reference management mechanisms grows with the amount of supported capacity. Our field experience, from a large number of deduplication product deployments, has shown that the cost of reference management (upon addition and deletion of data) has become one of the biggest real-world bottlenecks, involving operations that take many hours per day, and force a hard limit to scalability.

A lot of the research in the area has focused on optimizing deduplication efficiency and index management, without being able to sufficiently boost single-node capacity: with the current state-of-the-art a single node is limited to a few tens, or hundreds, of terabytes—which is far from sufficient for the petascale. Consequently, scalability has been addressed mostly through the deployment of complex, multi-node systems, that aggregate the limited capacity of each node in order to provide a few petabytes of storage at very high (acquisition, management, energy, etc.) cost. Surprisingly, the problem of reference management performance is largely ignored.

As the rate at which data are generated is rapidly increasing, the pressure for high-performance, scalable and cost-effective deduplication systems becomes more evident. We advocate that single-node performance is of key importance to next-generation deduplication systems: by making the most of a single node's resources, it is possible to build a high-performance deduplication system that will be able to scale to billions of objects. Based on our field experience, we know that such a system would be valuable to a very large number of users (e.g., small/medium businesses) where simplicity is also a top priority. Additionally, we believe that improving single-node performance is essential for multi-node systems as well, since a lot of our techniques can be used to provide more efficient building blocks for these systems, or even collapse them into a single node.

This paper presents a *complete*, single-node deduplication system that covers indexing, reference management, and end-to-end throughput optimization. We contribute new mechanisms to address dedupe challenges and combine them with well-known engineering techniques in order to design and evaluate the system considering all three dedupe goals. *Progressive sampled indexing* removes scalability limitations imposed by indexing, while serving most lookup requests in $O(1)$ time complexity from memory. Our index uses sampling to perform fine-grained indexing, and greatly improves scalability by requiring significantly less memory resources. We address the problem of reference management by introducing *grouped mark-and-sweep*, a mechanism that minimizes disk accesses and achieves near-optimal scalability. Finally, we present a modular, event-driven, client pipeline design that allows the client to make the most of its resources and process backup data at a rate that can fully utilize the dedupe server. As a result, our prototype can achieve high backup (1 GB/sec for unique data and 6 GB/sec for duplicate data) and restore throughput (1 GB/sec for single stream and 430 MB/sec for multiple streams) and good deduplication efficiency (97%), at high capacities (123 billion objects, 500 TB of data per 25 GB of system memory).

The rest of the paper is organized as follows: Section 2 gives a detailed description of the major challenges we had to address. In Section 3 we describe how we address them through our prototype's novel mechanisms, and in Section 4 we present our evaluation results.

2 Challenges

2.1 Indexing

Most deduplication systems operate at the sub-file level: a file or a data stream is divided into a sequence of fixed or variable sized *segments*. For each segment, a cryptographic hash (MD5, SHA-1/2, etc.) is calculated as its *fingerprint* (*FP*), and it is used to uniquely identify that particular segment. A *fingerprint index* is used as a catalog of FPs stored in the system, allowing the detection of duplicates: during backup, if a tuple of the form < FP, location_on_disk > exists in the index for a particular FP, then a reference to the existing copy of the segment is created. Otherwise, the segment is considered new, a copy is stored on the server and the index is updated accordingly. In many systems, the FP index is also crucial for the restore process, as index entries are used to locate the exact storage location of the segments the backup consists of.

The index needs to have three important properties: 1) scale to high capacities, 2) achieve good indexing throughput, and 3) provide high duplicate detection rate—i.e., high deduplication efficiency. Table 1 demonstrates how these goals become very challenging for a

Item	Scale	Remarks
Physical capacity C	C = 1,000 TB	
Segment size S	S = 4 KB	
Number of segments N	N = 250*10^9 segs	$N = C/S$
Segment FP size E	E = 22 B	
Segment index size I	I = 5,500 GB	$I = N * E$
Disk speed Z	400 MB/sec	
Block lookup speed goal	100 Kops/sec	Z/S

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

Petascale system. If the system capacity is 1 PB, and the segment size is 4 KB (for fine-granularity duplicate detection), indexing capacity will need to be at least 5,500 GB to support all 250 billion objects in the system. Such an index is impossible to maintain in memory Storing it on disk, however, would greatly reduce query throughput. To achieve a rate of 400 MB/sec, would require the index—and the whole dedupe system for that matter—to provide a query service throughput of at least 100 Kops/sec. Trying to scale to 1 PB by storing the index on disk would make it impossible to achieve this level of performance[1]. Making the segment size larger (e.g., 128 KB) would make deduplication far more coarse and severely reduce its efficiency, while still requiring no less than 172 GB of RAM for indexing.

It becomes obvious that efficient, scalable indexing is a hard problem. On top of all other indexing challenges, one must point out that segment FPs are cryptographic hashes, randomly distributed in the index. Adjacent index entries share no locality and any kind of simple read-ahead scheme could not amortize the cost of storing index entries on disk.

2.2 Reference Management

Contrary to a traditional backup system, a dedupe system shares data among files by default. Reference management is necessary to keep track of segment usage and reclaim freed space. In addition to scalability and speed, reliability is another challenge for reference management. If a segment gets freed while it is still referenced by files, data loss occurs and files cannot be restored. On the other hand, if a segment is referenced when it is actually no longer in use, it causes storage leakage.

Previous work [12, 19] mainly focused on indexing and largely ignored reference management. Some recent work [4, 18] started to acknowledge the difficulty of the problem. But, for simplicity, only simple reference counting was investigated without considering reliability and recoverability. Reference counting, however, suffers from low reliability, since it is vulnerable to lost or repeated updates: when errors occur some segments may be updated and some may not. Complicated transaction rollback logic is required to make reference counts consistent. Moreover, if a segment becomes corrupted, it is important to know which files are using it so as to recover the lost segment by backing up the file again. Unfortunately, reference counting cannot provide such information. Finally, there is almost no way to verify if the reference count is correct or not in a large dynamic system. Our field feedback indicates that power outages and data corruption are really not that rare. In real deployments, where data integrity and recoverability directly affect product reputation, simple reference counting is unsatisfactory.

Maintaining a reference list is a better solution: it is immune to repeated updates and it can identify the files that use a particular segment. However, some kind of logging is still necessary to ensure correctness in the case of lost operations. More importantly, variable length reference lists need to be stored on disk for each segment. Every time a reference list is updated, the whole list (and possibly its adjacent reference lists—due to the lists' variable length) must be rewritten. This greatly hurts the speed of reference management.

Another potential solution is mark-and-sweep. During the mark phase, all files are traversed so as to mark the used segments. In the sweep phase all segments are swept and unmarked segments are reclaimed. This approach is very resilient to errors: at any time the process can simply be restarted with no negative side effects. Scalability, however, is an issue. Going back to the example of Table 1, we would need to deal with N = 250 billion segments. If a segment FP is E = 22 bytes, that would be $I = N * E$ = 5,500 GB of data. If we account for an average deduplication factor of 10 (i.e., each segment is referenced by 10 different files), the total size of files that need to be read during the mark phase will be 55,000 GB. This alone will take almost 4 hours on a 400 MB/sec disk array. Furthermore, marking the in-use bits for 250 billion entries is no easy task. There is no way to put the bit map in memory. Once on disk, the bit map needs to be accessed randomly multiple times. This also takes significant amount of time. One might want to mitigate the poor performance of mark-and-sweep by doing it less frequently. But in practice this is not a viable option: customers always want to keep the utilization of the system close to its capacity so that a longer history can be stored. With daily backups taking place, systems rarely have the luxury to postpone deletion operations for a long time. In our field deployment, deletion is done twice a day. More than 4 hours in each run is too much. In a large production-oriented dedupe system reference management needs to be very reliable and have good recoverability. It should tolerate errors and always ensure correctness. Although mark-and-sweep provides

[1] Our measurements show that even high-end SSDs cannot achieve more than 60 Kops/sec

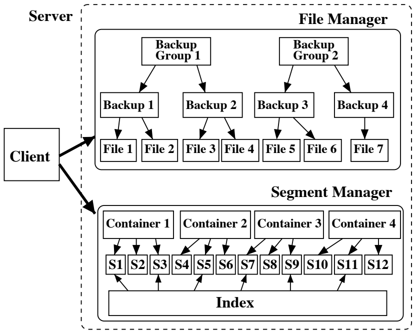

Figure 1: Client and deduplication server components. The server components may be hosted on the same or different nodes.

these properties, its performance is proportional to the capacity of the system, thus limiting its scalability.

2.3 Client-Server Interaction

Even if we solve the indexing and reference management problems, high end-to-end throughput is not guaranteed. An optimized client-server interface is necessary to reap the benefits of deduplication. The typical dedupe client performs the following steps during backup: 1) read data from files, 2) form segments and calculate FPs, 3) send FPs to the server and wait for index lookup results, and 4) for each index miss, transmit the relevant data to the server—otherwise create references to the existing segments. This process may suffer from three different types of bottlenecks. First, reading files from disk is an I/O-bound operation. Second, calculating cryptographic hashes is a very CPU-intensive task, and the client may not be able to compute FPs at the necessary rate. Finally, high latency and low communication throughput may become the main bottleneck for overall performance.

3 Prototype Design

3.1 Goals and System Architecture

We set our performance goals as follows:

- **Scalability**: store and index hundreds of billions of segments.
- **Deduplication efficiency**: best-effort deduplication: if resources are scarce, sacrifice some deduplication for speed and scale.
- **Throughput**: near-raw-disk throughput for data backup, restore, and delete.

To that end, we have implemented a prototype of our scalable duplication system aiming to validate the effectiveness of the proposed mechanisms. Our implementation uses C++ and pthreads, on 64-bit Linux, and it is based on the architecture shown in Figure 1.

The server side component consists of two main modules—the *File Manager* and the *Segment Manager*—that implement all the deduplication and backup management logic.

The File Manager (*FM*) is responsible for keeping track of files stored on the deduplication server. The FM manages file information using a three level hierarchy, visible in Figure 1. The bottom level consists of *files*, each represented by a set of metadata and identified by a file FP, calculated over all segment FPs that the file consists of. The middle level consists of *backups*, that group files belonging to the same backup session. At the top level, multiple backups are aggregated to a *backup group*, allowing the FM to perform coarse-granularity tracking of file/backup changes in the system, so as to assist our reference management mechanism.

The Segment Manager (*SM*) is responsible for the indexing and storage of raw data segments, and may run on the same or a different server than the FM. Segments are stored on disk in large (e.g., 16 MB) storage units, called *containers*. Containers consist of raw data and a catalog which lists all FPs stored in the container. All disk accesses are performed in the granularity of containers. Storing adjacent segments in the same container greatly improves dedupe performance, by reducing container I/O and by improving indexing efficiency (as discussed in Section 3.2.1). The SM also incorporates the dedupe index, and updates it when segments are added/removed.

The client component reads file contents or receives data streams (e.g., data from *tar*), performs segmentation, and calculates segment FPs. After querying the SM index, the client creates references to the existing copies of FPs located in the SM, and initiates data transfers for new FPs. Once a file has been fully processed, the File Manager is updated with file metadata.

Without loss of generality, we use fix-sized, 4 KB segments, for fine-granularity dedupe—although none of the mechanisms relies on this assumption.

3.2 Progressive Sampled Indexing

Most dedupe systems, when performing backup restore, rely on the index—or a similar catalog-like structure—in order to determine the disk location of each segment. This forces the strict requirement for at least one *complete index* containing location information for all FPs, that the system will have to maintain and protect against crashes, corruption etc., because errors cannot be tolerated. If a segment's disk location cannot be determined due to index failure, the whole file or backup gets corrupted. Maintaining such a data structure is a difficult and resource consuming task, that almost certainly impacts system scalability and performance, since the index typically needs to be stored both in memory, for performance, and on disk, for durability.

In order to address the indexing challenges and scale to billions of objects with high performance we had to remove this restriction by introducing *directly locatable objects*: when a file is stored in the system, file segment location information is stored with the file metadata, therefore removing the need to consult the index for the exact location of file segments. For example, if file F consists of segments with FPs A, B and C, stored at disk locations 1, 2 and 3 respectively, F would be represented by the list "$A,1,B,2,C,3$"—instead of just "A,B,C". The increased file metadata size is not a problem, since metadata are stored on disk, while the indexing freedom we get in exchange is extremely valuable.

By decoupling indexing and restore we no longer need to maintain a full index. Instead, we introduce *sampled indexing*, that is based on the observation that given certain amounts of memory and raw capacity, we can calculate the index size, and determine the number of entries that need to be dropped. In particular, if M is the amount of memory available for indexing (in GB), S is the dedupe segment size (in KB), E is the memory entry size (in bytes), and C is the total supported storage (in TB), then we can support M/E billion entries, while the system consists of a total of C/S billion segments. Therefore, if we assume a sampling period T, signifying that we maintain "1 out T" fingerprints in memory, we can define a sampling rate R as follows:

$$R = 1/T = (M/E)/(C/S) = (M*S)/(E*C) \qquad (1)$$

In the example of Table 1, using 22 bytes per index entry, with 4 KB segments and 64 GB of memory for indexing, we can support 11.6 TB of data with a sampling rate of 1 (i.e., a full index). Scaling to 1,000 TB, would require a sampling rate of 0.0116—i.e., insert in the index one out of 86 FPs. Using an 8 KB segment, we could double the raw capacity, or double the rate to $1/43$, sacrificing some dedupe accuracy for higher index density. Increasing the indexing capacity of the system by adding more RAM is rewarded with higher sampling rates (i.e., better dedupe efficiency), while increasing only the storage capacity results in a lower sampling rate, but this is often acceptable, in return for "infinite" system scalability.

3.2.1 *Dedupe efficiency: pre-fetching and caching.*

Since "1 out of T" FPs is inserted in the index, index hits—and, consequently, dedupe efficiency—would be reduced by a factor of T. However, when a lookup operation hits on a sampled FP (also referred to as a "hook"), we locate the container it belongs to and pre-fetch all FPs from that container's catalog into a memory cache. It has been shown [19] that the likelihood of subsequent lookups hitting on the FP cache is high, due to spacial locality: if hook FP A was followed by dropped FP B, then it is very likely that A and B will reappear in order in the future, in which case A will have seeded pre-fetching of its container catalog, resulting in a cache hit for B.

Container catalog pre-fetching can be extremely effective in improving the deduplication efficiency of a sampled index. However, pre-fetching introduces a minimum sampling rate: at least one FP per container (e.g., the first FP stored in the container) must be in the memory index as a hook, in order to seed pre-fetching. Because of this, if container size is K MB, then $R \geq R_{min} = S/(K*2^{10})$ and, subsequently, scalability is no longer "unlimited": the maximum supported capacity is now $C \leq (M*K*2^{10})/E$. For 4 KB segments and 16 MB containers, at least 1 out 4096 FPs needs to be sampled, and with 64 GB of RAM, as in the example of Table 1, $C \leq 47,662$ TB—which is still very high.

Deduplication efficiency. Although the combination of sampling and FP pre-fetching can often yield up to 100% duplicate detection, random eviction of cache entries may reduce deduplication. Using a simplified model we can estimate the dedupe efficiency of the system. Each container catalog contains at most $(K*2^{10})/S = 1/R_{min} = T_{min}$ entries. If we want to achieve deduplication efficiency f%, and we suffer x misses from one container, then:

$$f/100 = 1-(x/T_{min}) \Rightarrow x = T_{min}*(1-(f/100)).$$

If a particular container suffers one eviction during a large time frame (most likely scenario, especially when LRU is used), then all x misses will fall between two consecutive hooks hitting on the index, and therefore:

$$T = 1/R = x+1 \Rightarrow T = T_{min}*(1-(f/100))+1 \Rightarrow$$

$$\Rightarrow (E*C)/(S*M) = T_{min}*(1-(f/100))+1 \qquad (2)$$

Using Equation 2 we can calculate that in the example of Table 1, with 64 GB of memory, the deduplication efficiency will be $f = 97.9\%$. Alternatively, for a given target dedupe efficiency, we can calculate the necessary values to achieve it: for example, if we want $f \geq 95\%$, and given E, C and S, the amount of memory required is $M \geq 26.7$ GB.

3.2.2 *Progressive Sampling.*

A simple, yet important, optimization to sampled indexing is based on the observation that Equation 1 is using the total storage capacity of the system, and, therefore, calculates the value of R_{tot}, required to support all C/S billions of objects. However, at any given time, only the amount of data that are actually stored in the system need to be indexed, which allows us to utilize a *progressive* *sampling rate* that calculates R using the amount of storage *used*, as opposed to the maximum raw storage. Initially we set $R = 1$, and gradually decrease it as more storage gets used. In our working example, with 64 GB of RAM, $R = 1$ can index 11 TB of storage. As we approach the 11 TB limit, we can set $R = 0.5$ and down-sample the index (e.g., drop index FPs with $FP \ mod \ 2 \neq 0$), thus doubling the indexing capacity. Eventually, as usage approaches 1,000 TB, R will converge to $R_{tot} = 0.0116$.

3.2.3 Implementation

The index and cache have been implemented in C++ using a highly parametrizable hash table design, which we call *dhash*, optimized for high performance and efficient memory usage. The M GB of memory available for indexing are divided to fixed size buckets (1 KB by default), allowing us to have a maximum of $Y = M/bucket\_size\_in\_KB$ millions of buckets. No pointers are used in a dhash structure, and all operations use offsets, allowing us to 1) perform custom memory management (bucket slab allocator), 2) get memory savings by replacing each 8-byte pointer with 6 bytes of offset data, and 3) make the dhash easily serializable (e.g., when checkpointing to disk at system shutdown).

If a dhash is used at the role of the index, we aim to accommodate as many sampled FP entries as possible. We utilize 2^b buckets for the hash table, where $b = log_2(Y * 2^{20}) - k$. The system parameter k determines the number of buckets reserved for collision handling. Each index entry contains a partial FP (since the b least significant bits of the FP are encoded in the hash table position), and the container number the FP belongs to. For simplicity we use 128-bit MD5 (which is not strong enough for production, but adequate for our testing purposes), leading to a typical entry size of 18 bytes[2]. Each index dhash also utilizes a Bloom filter, to avoid unnecessary lookup operations, which greatly improves performance.

A cache dhash is optimized mainly for performance: it will use all buckets for the hash table, and handle collisions by running a cache eviction algorithm. A cache dhash can employ one of three eviction policies when collisions for a particular bucket Q occur: *Immediate eviction* will empty Q, and consider all the containers of Q's previous entries as evicted from the cache. This policy is very fast since it performs lazy eviction of FPs, allowing for subsequent lookups to hit on those entries. On the downside, this policy penalizes multiple containers at once. *Eviction by threshold* is similar to immediate eviction, but the containers whose entries are being removed from Q will not be considered as evicted until a certain percentage of their total entries has been removed from all cache buckets. This imposes less of a penalty to containers with entries in Q, but may lead to poor deduplication if the threshold is high, since a particular container may not be pre-fetched even though many of its entries have been evicted. *Container LRU* will evict the entries of the least recently pre-fetched container. If that does not free up space in Q, the process is repeated. Although this is the policy that yields maximum dedupe efficiency, it is also the one with the most overhead. Our default policy is immediate eviction, which provides good deduplication efficiency, and performance only slightly lower than eviction by threshold.

In order to provide high dedupe efficiency after system reboots or crashes, we must ensure that a relatively recent index checkpoint is stored persistently[3]. Bucket change-tracking combined with our pointer-free implementation make checkpointing efficient (only a few seconds per checkpoint). Our current policy creates checkpoints every few minutes, and on system shutdown.

SSD indexing. Although sampling provides an efficient way around scalability restrictions imposed by memory limitations, we wanted to also provide a way to improve scalability even with modest amounts of memory, and without having to resort to very low sampling rates. To that end we have also implemented a (persistent) SSD-based version of our sampled index. Sampled fingerprints are stored on sorted SSD blocks and all available memory is used for three performance optimizations: 1) create an SSD summary data structure *SSD_sum*, 2) maintain a Bloom Filter for the SSD index, and 3) maintain an FP cache of pre-fetched containers—similar to that used for the memory index. The *SSD_sum* data structure keeps track of the first FP in each of the SSD's (sorted) blocks, thus allowing us to perform any lookup with at most one SSD block read: when a *lookup(X)* operation is performed, X may be found in the cache, or it may be found by reading the SSD block i, where $SSD\_sum(i) \leq X < SSD\_sum(i+1)$. The SSD index is read-only, eliminating the need for shared locking during accesses. All SSD index updates are cached and logged. Eventually, index updates are performed in batches (and with the SSD exclusively locked): for our 128 GB SSD a full update takes less than 9 minutes, and we can afford to update the SSD many times per day.

3.3 Grouped Mark-and-Sweep

The challenge in reference management, as discussed in Section 2.2, is to ensure reliability while ensuring that the reference management mechanism is also both scalable and fast enough to keep up with the backup speed. A mark-and-sweep approach is very reliable, but offers

[2] With a stronger 160-bit hash, the entry size becomes 22 bytes.

[3] Notice that even if we lose all index index entries, correctness is preserved.

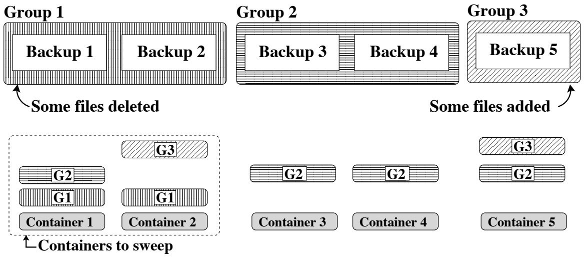

Figure 2: Example illustrating the scalability of grouped mark-and-sweep.

poor scalability because it needs to touch every file in the system. To address this challenge we propose the *grouped mark-and-sweep* (GMS) mechanism, which is reliable, scalable, and fast. The key idea is to avoid touching every file in the mark phase and every container in the sweep phase. GMS achieves scalability because its workload becomes proportional to the changes—instead of the capacity of the system.

The operation of GMS is based on change-tracking within the File Manager. As presented in Figure 1, the File Manager keeps track of files, backups, and backup groups. A file can be a regular file, a database backup stream, an email, etc. A backup is a set of files, e.g., all files under a set of directories. The creation and contents of backups are in the control of the user.

Backup groups aim to control the number of entries that GMS needs to manage, and are created and managed by the File Manager. When backups are small, we aggregate multiple small backups to one bigger backup group. The File Manager tracks changes to each backup group, and for each changed backup group, it further tracks whether files have been added to or deleted from it. During a GMS run, the following steps take place:

1. **Mark changed groups.** Only mark the changed backup groups and do nothing for unchanged backup groups. As an example in Figure 2, assume that File Manager's change tracking shows that, since the last GMS cycle, we deleted some files from group Group1, added some files to group Group3, and made no modifications to Group2. In this case we only need to touch files in backup groups Group1 and Group3. Usually, most backup groups (e.g., Group2) are not changed and files in those groups don't need to be marked. The mark results of G1 and G3 are recalculated by traversing all files in Group1 and Group3 and recalculating G1 and G3 for all containers that have segments used by those files. A group's mark results, say G1, is a bitmap implemented as a file for each container.
2. **Add affected containers to the sweep list.** Only containers used by groups that have deleted files need to be swept because only those containers may have segments freed. In the example of Figure 2, Group1 has files deleted and it has used containers 1 and 2. So we put these two containers in the sweep list. The segments in other containers are either still referenced by files in the unchanged groups (say Group2), or referenced by new files in new groups (say Group3).
3. **Merge, sweep, and reclaim freed space.** For each container in the sweep list, we merge the mark results of all groups using that container. If a segment is not used, it can be reclaimed. In the example of Figure 2, for Container 1, we merge (the old) G2 and (the new) G1, to determine potentially unused segments. Similarly, we merge (the new) G3 and (the new) G1, to determine potentially unused segments in Container 2.

As it becomes clear from the example of Figure 2, GMS provides two important scalability benefits. First, old mark results (e.g., G2) can be reused, without having to re-generate them in every mark-and-sweep cycle. Each set of mark results is stored and reused in the future, making the mark phase scalable by avoiding to touch the majority of the unchanged backup groups. Secondly, unlike conventional mark-and-sweep where all the entries are swept to determine the unused entries, in GMS we know which containers have reference removal operations, and the system only needs to sweep that subset of containers. Therefore the majority of containers in the system are usually not touched in the sweep step.

One drawback of GMS is that a group needs to be remarked even if just one file has been deleted from it. Fortunately the overhead is surprisingly small: segments can be marked at a rate of 26 GB/sec. Since most bitmaps are not changed, there are little work in the sweep phase.

Overall, GMS makes mark-and-sweep scalable by only touching the changed objects, while maintaining the reliability of mark-and-sweep. If errors occur, the whole process can start over and all operations are idempotent. Finally, the mark results (e.g., G1 and G2 for Container 1) serve as a coarse reference list for segments in the containers. When data corruption occurs in a container, the mark results can give us a complete list of backup groups that use that particular container. This limits the set of affected files significantly, and greatly enhances recoverability. Otherwise, we would need to go through all files in the system to determine which files are using that container.

Discussion. An interesting issue related to reference management is concurrent reference updates (data deletion) and data backup. In the example of Figure 2, Backup 5 may still be active when it gets marked, and after all changed backup groups are marked, GMS determines that segment x can be deleted. If Backup 5 uses

x between the time Backup 5 was marked and the time that GMS deleted segment x, data loss will occur as a backup uses deleted/non-existent segments. HYDRAstor [4] uses a read-only phase to freeze the system while updating segment reference counts. In practice, the viability is dubious. On a busy system, there are always some active backups. It is very unlikely to find a time window when the system can be frozen.

Our system uses an in-memory protection map to address this problem: after GMS begins, all segments used by current active backups are protected by storing their segment fingerprints in a protection map in memory. GMS only deletes segments whose fingerprint is not in the protection map. This way GMS can be certain that segments in use will never get deleted. The protection map grows while GMS is running and gets deleted once GMS completes. This is another reason why GMS needs to be fast enough to prevent the protection map from using too much memory. To mitigate the time spent in GMS, and limit the growth of the protection map, GMS can be done more frequently.

3.4 Client-Server Interaction

Even with high-performance server components, it is impossible to achieve high throughput, unless the client is able to push data to the server at a high-enough rate. To that end, our client component is based on an event-driven, pipelined design, that utilizes a simple, fully asynchronous RPC implementation.

Our RPC protocol is implemented via message passing over TCP streams or system IPC mechanisms (e.g., named pipes), depending on whether communication is remote or local. The TCP implementation utilizes multiple TCP connections to keep up with the throughput requirements. All RPC requests are asynchronous and batched in order to minimize the round-trip overheads and improve throughput. A client can register different callback functions for each type of RPC. The callback functions are used to deliver the RPC results to the caller as they become available.

Based on our asynchronous RPC protocol, we have implemented an event-driven client pipeline, presented in Figure 3, where each backup step is implemented as a separate pipeline stage.

First, the reader thread R receives the backup schedule, reads large chunks of data (e.g., 256 segments), and enqueues requests to the hash queue HQ. The hashing thread H dequeues requests from HQ, performs segmentation for each data chunk, and calculates FPs. Calculating cryptographic hashes is a computationally expensive operation, and, in order to fully utilize multiple CPU cores, H employs n MD5 worker threads $(H_1, H_2, \ldots, H_n)$ that calculate FPs asynchronously. Once a chunk's segment FPs have been calculated, callback function $CB1$ enqueues the updated request to the lookup queue LQ.

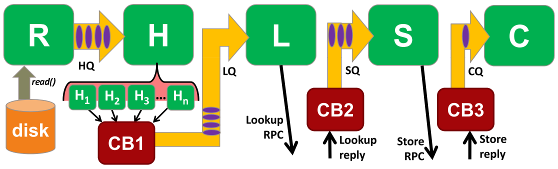

Figure 3: Client pipeline, consisting of five main event-handling threads connected using queues.

The lookup thread L receives requests from LQ and issues one single, batched, asynchronous lookup RPC to the server, incurring a single RPC round-trip for all 256 FPs. Callback function $CB2$ delivers the RPC reply and creates references to the containers of the FPs that were found on the server. If one or more FPs were not found, CB2 enqueues the updated request in the store queue SQ.

The store thread S receives requests from SQ, and sends raw data blocks to the back-end through one single, batched, asynchronous RPC. Callback function $CB3$ ensures that the write operation was successful, and forwards the last request for each file to the close queue CQ.

Finally, close thread C, receives the final request from CQ, performs cleanup, calculates file metadata, and updates the File Manager.

Client queues allow us to better understand system behavior. For instance, on a client with low hash calculation throughput, we can observe HQ to be full most of the time, while low network performance will lead to LQ and SQ being mostly full. In such cases, more than one threads can be used for each pipeline stage. By using two store threads, for example, we can consume requests from SQ at a higher rate.

4 Evaluation

Our main test-bed is an 8-core Xeon E5450 at 3 GHz with 32 GB RAM, running Linux. Our 24 TB disk array consists of 12 disks, 2 TB each, and uses RAID 0[4] to stripe all physical disks to a single logical volume.

We used two main data sets for testing. Our synthetic data set consists of multiple 3 GB files, each with globally unique data segments. Our second data set consists of virtual machine images, which are a very common real-world enterprise use-case, that takes advantage of deduplication. We use a VMware "gold" disk image ($VM0$), hosting a Microsoft Windows XP installation, and created three additional versions of it ($VM1$, $VM2$, and $VM3$), each with incremental changes: VM1 is VM0 with all Microsoft updates and service packs, VM2 is

[4] RAID 0 is not recommended for a high-availability system, but we used it to achieve maximum performance and mitigate the disk bottleneck—thus emulate a high-end array.

VM1 with a large anti-virus suite installed, and VM3 is VM2 after the installation of various utilities (document readers, compression tools, etc.). This data set aims to measure the "real-world" dedupe performance of our system, using a file type of great importance for the enterprise.

For both data sets we configured the system to use a sampling rate of $R = 1/101$, which is low enough to stress the system. For the synthetic tests performed on our current test-bed, the index uses 25 GB memory to hold 1.23 billion FPs. With a sampling rate of 1/101, this is equivalent to a full index of 124 billion FPs, or 500 TB of raw storage—given that our segment size is 4 KB[5].

4.1 Throughput

4.1.1 Backup Throughput

Index throughput. Before performing any macro-benchmarks, we used micro-benchmarks to ensure that the index can support our goals—e.g., in the example of Table 1, at least 400 MB/sec. In all the micro-benchmarks the index could easily handle the desired rates: insert/lookup/remove cost does not exceed 7,619/12,020/16,836 cycles, respectively, even when index occupancy is more than 97%. For instance, on a 3 GHz CPU, and in the worst-case scenario where all incoming FPs exist in the system (and the Bloom filter is of no help), the index can sustain a backup rate of around $975{*}T$ MB/sec, where T is the sampling period. For our test configuration, $T = 101$, and the index can sustain a rate of about 98.5 GB/sec.

Unique data: baseline vs. prototype. Figure 4 shows the backup throughput using the synthetic data set. We vary the number of concurrent backups, in steps of 1, 4, 16, 32, 64, 128 and 256, in order to evaluate the system's capability for concurrency. For consistency, all backups consist of multiple 3 GB files that add up to 768 GB.

The unique data throughput test aims to measure the prototype's behavior in the absence of duplicates. Unique data can be significant when a client performs the initial backup or a lot of changes have been made. This test stresses the disk and the network systems as large amounts of data need to be transferred.

To get a sense of the performance of raw hardware, we first measured a baseline throughput. The baseline throughput of the disk array ("Baseline" in Figure 4), is measured by writing the same synthetic workload to the file system. For a single backup, the baseline throughput is around 1 GB/sec. This is the maximum throughput of the storage system. The baseline throughput quickly drops to around 300 MB/sec for storing multiple backups concurrently because disk contention increases with the number of concurrent backups.

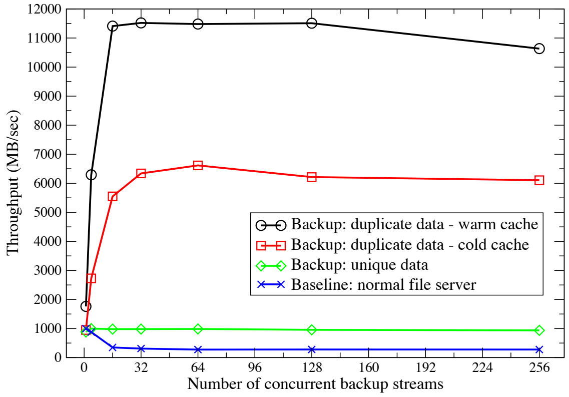

Figure 4: Aggregate throughput for our synthetic data set, with varying number of concurrent backups. Our system is capable of 6 GB/sec for duplicate data backup, and close to 1 GB/sec for concurrent backups of unique data. Dedupe efficiency is 97%, and we support 200 TB storage for every 10 GB of system memory (500 TB for 25 GB in this test).

Backing up the same data set ("Unique data" in Figure 4) using our prototype achieves a steady throughput of about 950 MB/sec as we scale to multiple concurrent backups, which is significantly better than the regular file server. This is mainly because our prototype performs segmentation on all incoming data, and manages the serialization of containers to disk (regardless of content source), therefore decreasing concurrent disk accesses.

Duplicate data backup throughput. After backing up the unique data workload using our prototype, we backup the same files again ("Duplicate data - cold cache" line in Figure 4). This time, all segments are duplicates, and we aim to observe how our prototype performs when it only needs to reference existing data, instead of physically storing new data. This test mainly stresses the index lookup and disk pre-fetching operations.

Initially, for low levels of concurrency, the penalty for small random disk reads, for container FP catalog pre-fetching, dominates performance. Throughput improves steadily as we increase the level of concurrency and duplicate elimination pays off, with aggregate disk throughput reaching over 6.6 GB/sec for 64 concurrent backup streams. When disk accesses are already random, concurrent access doesn't introduce more randomness. On the other hand, concurrent accesses can fully utilize every disks in the disk array. Thus the aggregate throughput increases. After 64 concurrent streams, the disk ar-

[5] Testing our system with a configuration that supports a raw capacity of 500 TB per node may seem inadequate at first. One should keep in mind, however, that 1) We are stressing the system by using 4 KB segments. Most systems use significantly larger segments, leading to higher raw capacities. 2) This is *single-node* capacity with only 25 GB memory for indexing. As such, it is higher than that of most systems we know of (as presented in Section 4.4). Unfortunately we don't have access to servers with more memory or larger disk arrays so as to test higher capacities.

Backup streams	Unique data	Duplicates (cold cache)	Duplicates (warm cache)
1	840 (-4.9%)	699 (-26.4%)	1,989 (12.9%)
4	992 (-0.5%)	2,556 (-6.3%)	6,326 (0.6%)
16	999 (1.9%)	4,802 (-0.2%)	11,992 (5.1%)
32	985 (0.3%)	6,420 (1.3%)	12,134 (5.3%)
64	984 (-0.2%)	6,621 (0.1%)	11,865 (3.3%)
128	988 (3.2%)	6,315 (1.6%)	11,755 (2.1%)
256	955 (1.9%)	6,041 (-1.1%)	11,946 (12.3%)

Table 2: We repeated the experiments of Figure 4 using the SSD index. Results are in MB/sec. The percentages in parentheses show how much faster/slower the SSD index is from the memory index.

ray's capacity for pre-fetching is saturated and mild effects from concurrency overhead (index/cache locking, disk accesses etc.) are becoming obvious: duplicate data backup throughput falls to 6 GB/sec and remains mostly constant.

To verify our conjecture that duplicate data backup throughput limitations are mainly due to disk bottleneck (container FP catalog pre-fetching) instead of CPU, we backup the same files a third time immediately after the second backup. In this case, many FPs are already in the cache and fewer disk pre-fetches will be necessary. The throughput is shown as "Duplicate data - warm cache" in Figure 4. First we observe that overall throughput is much higher, reaching 11.5 GB/sec at around 16 streams, confirming that the bottleneck in our previous tests was in the disk random access performance, which determines the duplicate backup throughput. Additionally, we observe that the effects of concurrency are barely visible: aggregate throughput is stable up to 128 concurrent backups, but at 256 concurrent streams the overhead of pthread shared locks used for protected accesses to the FP cache buckets, as well as a few cache evictions that render the cache less "warm", take their toll—slightly lowering the aggregate throughput (10.6 GB/sec).

SSD indexing throughput. Using SSD index implementation on an 128 GB SSD drive, we repeated the throughput experiments of Figure 4 in order to 1) test the efficiency of our SSD indexing design, and 2) verify the effects of shared locking to duplicate data backups—since the SSD index is read-only and uses no shared locks. For our tests, we maintained the same sampling rate ($R = 1/101$) and used the same amount of memory for caching as before (2 GB)—so as to make a fair comparison. Notice that with this setup we are now using a total of only 10 GB and the amount of raw storage the system can support rose from 500 to 1,600 TB. Due to our efficient SSD index design and the lack of shared locking, most throughput results were similar or superior to those of the memory index. Table 2 summarizes the results and difference between the SSD index and memory index throughput. Notice, however, that these results include the cost of updating the SSD every time 65,536 new sampled entries have accumulated. A less (more) frequent SSD update policy would yield faster (slower) throughput results.

CPU cores	Unique data	100% Duplicates (cold cache)	100% Duplicates (warm cache)
1	347	354	356
2	599	612	612
4	900	1,167	1,172
8	907	1,983	2,004
14	925	2,373	2,485

Table 3: End-to-end backup throughput using a varying number of CPU cores. All numbers in MB/sec.

End-to-end throughput. Our next test attempted to include client performance in our evaluation, in an end-to-end system test, using a single 25 GB backup stream of unique segments. As presented in Table 3, we varied the number of CPU cores dedicated to MD5 calculation, and performed three tests for each configuration: an initial backup, a second backup of the same data with cold caches, and a third run with warm caches. All backups were performed using a 16-core Intel Xeon E5520 "client", with 32 GB of RAM, running RedHat Enterprise Linux 5. The results of Table 3 show that backing up unique data does not get much faster with more than 4 cores. Careful observation revealed two reasons for this behavior. First, even when using the Linux loopback interface, we could not get throughput higher than 10 Gbps, on that particular host. Notice that when bulk data transfers become unnecessary, the performance reaches 2.49 GB/sec. Second, we realized that careful optimization of our simple RPC mechanism might be able to yield better performance. However, optimizing network behavior and the RPC implementation is beyond the scope of this study. In order to evaluate the real throughput of our client design we made the assumption of an infinitely fast network/RPC infrastructure, and, temporarily, eliminated the network performance bottleneck. This revealed the client's full potential: running on our (slower) main Intel Xeon 5450 server, the client was able to push 360/697/1,023/1,319 MB/sec of unique data, with 1/2/3/4 cores dedicated to MD5, respectively.

Backup throughput conclusions. In summary, our backup throughput experiments show that, when backing up unique data, our system is nearly as efficient as a normal file server for single stream backup (no penalty for deduplication) and several times faster for multi-stream backups. This shows that our system can better organize the data on disks to achieve high throughput even with concurrent backups. When data are mostly duplicates, we can achieve 950 MB/sec for single stream backup and 6 GB/sec for multi-stream backups. Multiple

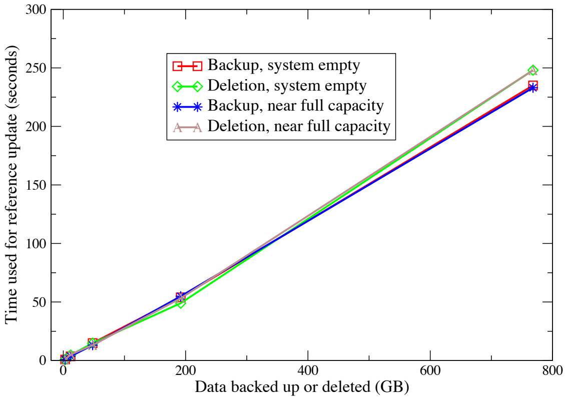

Figure 5: The reference update time for a given amount of data backed up or deleted when the system is empty and nearly full. The time is proportional to the data changed, and the slope shows the update throughput (3 GB/sec). Notice that the throughput is stable regardless of the capacity of the system or the amount of changed data.

streams help improve the aggregate throughput because they maximize the throughput of container FP catalog pre-fetching.

The major limitations that we observed are due to hardware restrictions: limited container pre-fetching throughput and CPU/networking bottlenecks in our end-to-end performance tests. On a production system equipped with hundreds of fast-seeking physical disks, and utilizing faster network connectivity, we expect to see much higher throughput. The only software limitation we observed was due to pthread locks, and is considered of secondary importance since it only impacts throughput minimally for more than 128 concurrent backup streams.

4.1.2 Reference Update Throughput

A critical property that is not often tested in deduplication systems, is the performance of reference updates, especially when we need to delete data—an operation that happens almost daily. Figure 5 shows reference update times measured when the synthetic data set gets backed up or deleted, both when the system is empty and near full capacity. The time is linear with the size of data backed up or deleted, because we need to update the reference of each segment that gets used.

The slope of the line corresponds to the throughput of the reference update, which is 3.2 GB/sec for data addition, and 3.1 GB/sec for data deletion. Deletion is slightly slower because when segments get deleted, they also need to be removed from the index. Contrary to a regular file-system, the deletion throughput of the deduplication system is slow because we pay the price of data sharing. However, it is still faster than the backup throughput of new data, which prevents the backup process from having to stall and wait for the deletion mechanism to free up space.

	Unique segs	Total unique	Ideal MBs	Real MBs	De-dupe
VM0	518,326	518,326	2,123	2,211	96%
VM1	733,267	921,522	3,775	3,938	96%
VM2	904,579	1,189,230	4,871	5,085	96%
VM3	1,145,029	1,616,585	6,621	6,860	97%

Table 4: Deduplication efficiency results for subsequent backups of four different versions of a Windows XP VMware image file.

4.1.3 Restore Throughput

Deduplication system benchmarks are dominated by backup testing and testing of restore is mostly ignored—probably because the restore process is usually slow, and correctness is the main concern. However, restore is an important operation and we wanted to ensure that our prototype provides sufficient performance. During our tests all data were restored correctly. Our single stream restore throughput was measured around 1 GB/sec, and 430 MB/sec for two or more concurrent restore streams. Single stream restore is fast because most accesses are sequential, while multiple concurrent restore streams introduce disk seeking. The use of directly locatable objects allows us to perform restore without using the index, making the whole process very scalable.

4.2 Deduplication Efficiency

Although we are willing to sacrifice some dedupe accuracy for high scalability, we still want to make sure the system provides adequate duplicate detection. In particular, since sampling provides the desired scalability, dedupe efficiency will be mostly determined by the effectiveness of pre-fetching.

In our synthetic data set, the true ("ground truth") duplication is 100%. Our prototype consistently eliminates no less than 97% of duplicates. This is consistent with the theoretical expectation, based on Equation 2: when we pre-fetch FPs from the container catalog, and because the sampling rate is 1 out of 101, the first 100 FPs may not be found. After the first hit, (101st FP in the worst case), we pre-fetch all FPs in that container. So theoretically we may fail to detect 100 over 4096 FPs, i.e., 2.4%.

For our VMWare data set we used our test sampling rate of 1/101, and a small FP cache (256 MB) in order to ensure that the cache cannot hold the whole working set. We performed multiple backups of each VM image, observing 100% dedupe efficiency for each run, with very high throughput (2.4 GB/sec). A more interesting experiment, however, presented in Table 4, is the dedupe efficiency achieved when backing up VM0, VM1, VM2, and VM3 back-to-back. Image VM0 has 518,326 4 KB segments, taking up 2,211 MB of disk space, instead of 2,123 MB, giving us 96% of the ideal dedupe efficiency. Backing up VM1 introduced 403,196 new segments (330,071 of VM1's segments were also in VM0), taking up 3,938 MB, for a steady dedupe efficiency of 96%. Similarly, VM2 and VM3 were deduplicated at 96% and 97% of the optimal dedupe rate, which is a very satisfying result for a cache of only 256 MB. These results are particularly encouraging, since field experience has demonstrated that VM image backups are one of the most common and effective uses of dedupe.

4.3 Scalability

In order to test the scalability of the system we first populated it to near-full capacity (480 out of 500 TB i.e., 95.5%) with unique data. Because our disk array is only 24 TB, we stored everything except the actual segment data. As the code mainly operates on the metadata, discarding segment data has no impact on the correctness of the test. After the system was populated we repeated the same throughput tests, during which everything was stored on disk (including segment data).

Figure 6 presents a throughput comparison between an empty and near-full system. For multi-stream throughput, the system occupancy has negligible performance impact because for both unique and duplicate data the throughput is, once again, bounded by the disk's sequential write and random read performance, respectively. When the system is near full capacity, the index lookup and update time increase slightly. But the main bottleneck is still disk I/O—overshadowing the effects of CPU overhead. This means that the throughput of the system will scale well in terms of system capacity while disk I/O is the main bottleneck—which is probably going to be true in the foreseeable future.

The index overhead does show up for single stream throughput. The throughput of single stream backup near full capacity is slower than that of the empty system because single stream throughput is CPU bound and accessing a "fuller" index takes a little bit more CPU time.

Figure 5 also compares reference update performance when the system is empty and near-full. As expected, the time for reference update is almost the same, since the grouped mark-and-sweep algorithm only touches the changed backup groups. The majority of the references, regardless of how many they are when the system is near full capacity, are not touched by the grouped mark-and-sweep. Finally, we also checked the deduplication efficiency for both the synthetic and real data sets and observed no degradation in a near-full system.

Our results demonstrate that all parts of our prototype are able to scale to high capacity, with almost no performance decrease. We are confident that our system would scale to higher capacities, given more resources. Moreover, the raw capacity supported by our system (200 TB for every 10 GB of memory) is higher than the capacity of any other single-node system presented in Section 4.4.

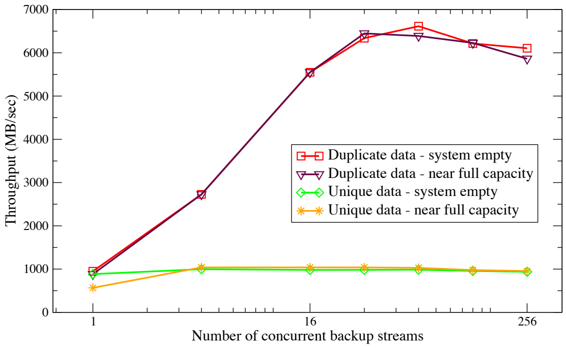

Figure 6: Throughput scalability tests show that there is no significant throughput drop when we get close to full capacity. We incur O(1) cost for most index operations, and throughput is disk-bound for both unique and duplicate data backups.

4.4 Comparison to State-of-the-art

When evaluating dedupe systems it is often the case that custom methods and private workloads are used to quantify the effectiveness of the proposed mechanisms (e.g., [19] and [12]). Comparisons to other systems are usually difficult, and limited to references to results reported by vendors, mostly because there is no agreed deduplication benchmark that would make benchmarking and comparisons fair and meaningful. Furthermore, when aiming to top the performance of state-of-the art systems, it is almost impossible to justify the cost and effort of obtaining, deploying and benchmarking even a single one of them. In our evaluation we tried to use data sets that will exercise the system in interesting ways, and that are relatively easy to be recreated and tested by other systems.

Table 4.4 presents some of the most popular high-performance deduplication solutions available as of April 2011. Assuming that all systems provide adequate deduplication efficiency (specifications do not provide precise numbers), we can see that our prototype's peak performance is similar to or better than that of all systems, with the exception of NEC's HydraStor. However, notice that HydraStor utilizes a large distributed system (55 "accelerator" and 110 "storage" nodes) in order to achieve its maximum throughput, and yet its raw capacity is limited to only 1.3 PB. Our prototype's single-node scalability competes with that of all systems and surpasses most of them, especially considering the limited amount of resources we have used (e.g., only 25 GB of RAM per 500 TB for $R = 1/101$, on an older 8-core server). Notice, however, that our goal to increase single-node scalability is not meant to replace all multi-node systems, but to potentially improve them by enabling each node to make better use of its resources and increase

Product	Backup (MB/sec)	Capacity (TB)	Nodes
DataDomain DD890 [3]	4,083	384	1
HP D2D4324 [7]	1,110	18	1
IBM ProtecTier [8]	1,000	1,000	2
Greenbytes GB4000 [6]	950	216	1
NEC HydraStor HS8-3110R [14]	41,250	1,300	55 + 110
Our prototype	6,000	500	1

Table 5: Summary: state-of-the art dedupe products as of April 2011.

data density per node. By doing so we could decrease the number of nodes necessary for a particular deployment, thus significantly decreasing the overall (acquisition, management, energy, etc) cost.

5 Related Work

Since the days of early deduplication systems, that performed mostly file-level or naive block-level deduplication [1, 11, 16], a lot of effort has been put into optimizing duplicate detection. In particular, many systems have investigated methods to perform content-aware segment boundary calculation, aiming to improve better duplicate coverage. Any degradation in dedupe efficiency was considered unacceptable. Such variable-size segmentation algorithms, utilize different variations of byte-level approaches, such as sliding window approaches (e.g., [5]), rolling hashes (e.g., [15]), Rabin fingerprints [2], and bimodal chunking [10]. For instance, systems like MAD2 [18], HYDRAstor [4, 17], as well as deduplication solutions by DataDomain [19] and Hewlett-Packard [12], utilize variable-size segments, in an attempt to achieve maximum compression. However, even if these algorithms make the best of raw storage (which is not always the case, as observed by [9]), single-node capacity is limited. Our work takes a different approach: we are willing to sacrifice some deduplication efficiency in order to achieve higher single-node scalability.

A sampling method is used in [12] to address indexing scalability restrictions. However, that approach is significantly different from ours, since it uses sampling to probabilistically identify "super-segments" that are used to perform coarse-granularity deduplication. Our segmentation algorithm operates at fine granularity at all times, and sampling is not used for pattern-matching, but for indexing actual file segments. Additionally, our approach is significantly more scalable, and can operate under heavy memory constraints, with good sampling rates: in a setting similar to the experiments presented in [12], our sampled index would require about 74% less memory (4.4 GB instead of 17 GB, with $R = 1/101$).

A lot of systems have used spacial locality to perform some type of caching (e.g., [18, 19]), but, to our knowledge, it has not been used before in combination with an aggressive sampling approach, such as the one we are proposing.

Our key assumption difference from previous efforts is that we are willing to relax our duplicate detection efficiency requirements, in order to address *all three* major challenges of single-node deduplication. Most other systems have provided good solutions for a subset of problems, usually excluding single-node scalability and reference management. For instance, DataDomain [19] addressed the disk bottleneck, by introducing a series of optimizations, including a Bloom filter, and spacial locality. However, their system can support a limited amount of raw storage, and is limited by network performance, since duplicate detection is performed only at the server. Additionally, it is not clear whether DataDomain's system can perform truly scalable resource reclamation.

HYDRAstor [4] on the other hand, achieves good scalability, but it does so by using a highly distributed, hierarchical model, with each node holding only a few tens of TB of storage. This design yields a high backup throughput, but at the cost of a highly distributed, costly system. Deletions in HYDRAstor, are implemented with a distributed reference counting method, which is difficult to maintain correctly, and scale without a large performance hit.

MAD2 [18] also uses a distributed storage system to provide scalability, as well as a number of optimizations that include spacial locality caching, and Bloom filters. Deletions are a very challenging operation in this system as well: they are performed only at the file level, and they are also handled by a variant of reference counting, with all the scalability and correctness problems discussed in Section 2.2. To our knowledge, our grouped mark-and-sweep approach is the only truly scalable, documented reference management implementation, that is also very resilient to errors.

Many scalable systems have adopted the event-driven design, however it is interesting that the nature of our application requires that we utilize it for the *client*, rather than the server. A pipelined client design was also proposed by [13], but it is significantly different from our design: it assumes pipeline stages whose operation requires a fixed amount of time, making it unrealistic for network operation. It also uses disk-based, client-side indexing, it implements a lot of functionality in the kernel, and it achieves scalability and throughput that is orders of magnitude lower than those of our client design.

6 Conclusion

Important engineering challenges need to be addressed in order to achieve the scalability, throughput and deduplication efficiency necessary to provide next-generation deduplication support. We have presented a clean-slate design that aims to maximize overall single-node effectiveness, and introduces new mechanisms that address the most pressing of these challenges. Our directly locatable objects enable the use of progressive sampled indexing—in memory or on SSD—which provides superior single-node scalability and memory usage efficiency—unlike any other system we know of. Our grouped mark-and-sweep mechanism attacks the difficult, and often neglected, resource management and reclamation problem, in a truly scalable and efficient manner. Additionally, we have proposed an asynchronous interface to the server back-end, capable of pushing data to the server at a high-enough rate.

The performance of our prototype validates the effectiveness of our design. Progressive sampled indexing achieves very good deduplication efficiency, while using only 10 GB of memory per 200 TB of raw storage (25 GB for 500 TB in our tests). Additionally, we were able to achieve backup throughput ranging from 950 (all unique data) to 6,000 MB/sec (all duplicate data), with deduplication efficiency no less than 97%, while our grouped mark-and-sweep approach can process data with speeds higher than 3.1 GB/sec, demonstrating that single-node dedupe effectiveness can be greatly improved by making good use of available resources.

Acknowledgments

The authors would like to thank Weibao Wu, Joe Pasqua, Sanjay Sawhney, Kent Griffin, Tzi-cker Chiueh, Vish Janakiraman, Vic Pantaleon, and the Symantec PureDisk team for their input and their help on evaluating the system. We would also like to thank the anonymous reviewers for their valuable comments.

References

[1] Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. Farsite: federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev. 36*, SI (2002), 1–14.

[2] Broder, A. Z. Some applications of rabin's fingerprinting method. In *Sequences II: Methods in Communications, Security, and Computer Science* (1993), Springer-Verlag, pp. 143–152.

[3] DataDomain. EMC data domain dd890. `http://bit.ly/eXL6Uc`.

[4] Dubnicki, C., Gryz, L., Heldt, L., Kaczmarczyk, M., Kilian, W., Strzelczak, P., Szczepkowski, J., Ungureanu, C., and Welnicki, M. Hydrastor: a scalable secondary storage. In *FAST '09: Proccedings of the 7th conference on File and storage technologies* (Berkeley, CA, USA, 2009), USENIX Association, pp. 197–210.

[5] Forman, G., Eshghi, K., and Chiocchetti, S. Finding similar files in large document repositories. In *KDD '05: Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining* (New York, NY, USA, 2005), ACM, pp. 394–400.

[6] GreenBytes. GB-X Series. `http://bit.ly/fTOSXd`.

[7] HP. StorageWorks D2D Backup Systems. `http://bit.ly/bfhyiU`.

[8] IBM. ProtecTIER Deduplication Solutions. `http://bit.ly/dVg3Z5`.

[9] Jin, K., and Miller, E. L. The effectiveness of deduplication on virtual machine disk images. In *SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (New York, NY, USA, 2009), ACM, pp. 1–12.

[10] Kruus, E., Ungureanu, C., and Dubnicki, C. Bimodal content defined chunking for backup streams. In *FAST'10: Proceedings of the 8th USENIX conference on File and storage technologies* (Berkeley, CA, USA, 2010), USENIX Association, pp. 18–18.

[11] Kulkarni, P., Douglis, F., LaVoie, J., and Tracey, J. M. Redundancy elimination within large collections of files. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2004), USENIX Association, pp. 5–5.

[12] Lillibridge, M., Eshghi, K., Bhagwat, D., Deolalikar, V., Trezise, G., and Camble, P. Sparse indexing: large scale, inline deduplication using sampling and locality. In *FAST '09: Proccedings of the 7th conference on File and storage technologies* (Berkeley, CA, USA, 2009), USENIX Association, pp. 111–123.

[13] Liu, C., Xue, Y., Ju, D., and Wang, D. A novel optimization method to improve de-duplication storage system performance. In *ICPADS '09: Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 228–235.

[14] NEC. HYDRAstor HS8-3000 Series. `http://bit.ly/hCHxhn`.

[15] NetApp. Deduplicating Backup Data Streams with the NetApp VTL, 2009. `http://bit.ly/buXcaV`.

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

[17] Ungureanu, C., Atkin, B., Aranya, A., Gokhale, S., Rago, S., Calkowski, G., Dubnicki, C., and Bohra, A. Hydrafs: a high-throughput file system for the hydrastor content-addressable storage system. In *FAST'10: Proceedings of the 8th USENIX conference on File and storage technologies* (Berkeley, CA, USA, 2010), USENIX Association, pp. 17–17.

[18] Wei, J., Jiang, H., Zhou, K., and Feng, D. Mad2: A scalable high-throughput exact deduplication approach for network backup services. In *MSST'10: Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies* (2010), pp. 1–14.

[19] Zhu, B., Li, K., and Patterson, R. H. Avoiding the disk bottleneck in the datadomain deduplication file system. In *FAST '08: Proceedings of the Conference on File and Storage Technologies* (2008), pp. 269–282.