Concurrent Deletion in a Distributed Content-Addressable Storage System with Global Deduplication

Przemyslaw Strzelczak, Elzbieta Adamczyk, Urszula Herman-Izycka, Jakub Sakowicz, Lukasz Slusarczyk, Jaroslaw Wrona and Cezary Dubnicki

9LivesData, LLC

{strzelczak, adamczyk, izycka, sakowicz, slusarczyk, wrona, dubnicki}@9livesdata.com

Abstract

Scalable, highly reliable distributed systems supporting data deduplication have recently become popular for storing backup and archival data. One of the important requirements for backup storage is the ability to delete data selectively. Unlike in traditional storage systems, data deletion in distributed systems with deduplication is a major challenge because deduplication leads to multiple owners of data chunks. Moreover, system configuration changes often due to node additions, deletions and failures. Expected high performance, high availability and low impact of deletion on regular user operations additionally complicate identification and reclamation of unnecessary blocks.

This paper describes a deletion algorithm for a scalable, content-addressable storage with global deduplication. The deletion is concurrent: user reads and writes can proceed in parallel with deletion with only minor restrictions established to make reclamation feasible. Moreover, our approach allows for deduplication of user writes during deletion. We extend traditional distributed reference counting to deliver a failure-tolerant deletion that accommodates not only deduplication, but also the dynamic nature of a scalable system and its physical resource constraints. The proposed algorithm has been verified with an implementation in a commercial deduplicating storage system. The impact of deletion on user operations is configurable. Using a default setting that grants deletion maximum 30% of system resources running the deletion reduces end performance by not more that 30%. This impact can be reduced to less than 5% when deletion is given only minimal resources.

1 Introduction

A scalable secondary storage keeps backups for a large number of protected systems. Backup servers write concurrently many backup streams (full/incremental/differential) and delete specified backups according to implemented backup retention policies. Backup applications access such system usually with a standard protocol like NFS or CIFS; these applications are often not aware of special functionality provided by the storage system like deduplication.

Secondary storage for enterprise market needs to offer key features like large capacity, high performance and high reliability. Since backup involves storage of multiple versions of similar data, deduplication is also naturally a desired functionality of such systems. With deduplication, logical storage capacity is far larger than physical space available resulting in substantial savings. Moreover, storing highly duplicated backup streams is usually much faster than writing the same data without deduplication. As a result, powerful scalable deduplication systems [9, 10, 12, 13, 14, 16, 21, 28] deliver shortened backup windows which is of primary importance to enterprise customers.

In this work we present a deletion algorithm designed for scalable deduplicating systems, in which deduplication is delivered with a content-addressable storage (CAS) using variable size blocks. This approach is quite popular today in commercial systems [10, 12, 13, 14]. We assume that block references as pointers to other blocks are made explicit in the storage. Deleting data in such system is in a way similar to the traditional concurrent garbage collection. However, in our case the deletion is significantly complicated by deduplication. For example, a simple solution of disabling deduplication when deletion is running may increase space consumption considerably, which may be not acceptable. Far better approach is to allow deduplication all the time, even concurrently with deletion. Desired system scalability, failure tolerance and support for dynamic reconfiguration further complicate the deletion for such systems.

This paper makes the following contributions. First, it identifies requirements for deletion in a distributed block store supporting deduplication. Second, we present an

algorithm satisfying these requirements and its implementation in a commercial system HYDRAstor [10] (in particular, this new algorithm allows for writing with deduplication during deletion, whereas the one described in [10] required a read-only period to perform deletion). Third, this work discusses the results of an evaluation of the deletion procedure, demonstrates its efficiency and illustrates impact of the deletion on user operations.

The remainder of this paper is organized as follows. Section 2 discusses deletion requirements. The challenges for deletion introduced by CAS with deduplication are identified in Section 3. Section 4 defines user-visible data model and proposed semantics of deletion which allows for active deduplication during deletion. The deletion algorithm implementing this semantics in a centralized storage is discussed in Section 5. Section 6 introduces a CAS-based distributed system and discusses modifications of our deletion algorithm for such architecture. Section 7 describes how the implemented deletion addresses the requirements identified earlier. The implementation of our deletion algorithm in a commercial system and the resulting performance are discussed in Section 8. Related work is given in 9. We conclude and discuss future work in Section 10.

2 Requirements on deletion

The deletion in all systems must first of all let users delete data selectively and preserve data which users decided to keep. Usually, once data is marked for deletion, space reclamation is done immediately to ensure high storage utilization. However, for a secondary storage supporting deduplication, space reclamation can be delayed because such storage is already highly space-efficient due to deduplication. Moreover, immediate retrieving of unused space in such systems could require significant resources like CPU and disk spindles, whereas delayed reclamation of unused space in batches can be implemented efficiently.

In enterprise backup system we need to minimize the impact of deletion on user operations such as backup, restore and replication. This functionality is expected to be available 24x7 and its performance should not suffer much from deletion running. Limiting running deletion to read-only periods is difficult to accept as it may result in insufficient backup windows. During deletion, deduplication of data being written by clients should be continued, as disabling it could lead to system becoming full and stopping user backup.

Additionally, the impact of deletion on user operations should scale proportionally with the whole system.

With an increasing number of machines and disks in the system, the probability of hardware failures increases. To ensure system availability, the deletion process must be able to continue its operation in the presence of multiple disk, node and network failures. Moreover, since ability to delete data is critical for operation of the system, the deletion process should be able to re-start and finish even in case of permanent data failure; that is why resiliency of deletion should be in fact higher than resiliency of user data.

Last but not least, the deletion should not require additional significant storage to complete.

3 Challenges for deletion introduced by content-addressable storage

Standard garbage collection techniques are not easily applicable to the problem of deleting data in a CAS-based system with deduplication.

3.1 Simplified model of CAS-based system supporting deduplication

A CAS-based storage comes with an interface available for its clients. A backup application usually does not use this interface directly, instead it is used by a software driver to implement standard protocols like NFS or CIFS, as depicted in Figure 1(a). A basic data unit used in this interface is a *block*. Besides binary data, blocks can contain *pointers* to other blocks which are those block addresses. A *block address* is derived from the block content, including pointers to other blocks if there are any. Content-addressing implies that blocks are immutable, as otherwise their addresses would change. However, the metadata associated with each block is mutable.

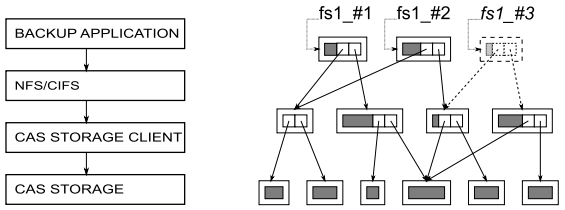

Figure 1: (a) System architecture. (b) User-visible data model (shaded rectangles represent data, white - block addresses). The first two backups of fs1 have been finished. Root of the third backup has not been written yet.

Because of deduplication, an address returned by a block write can refer to an already existing block called a *base block* for this duplicate write.

By writing blocks, a client can create a graph of blocks. Since blocks are immutable, such graphs are acyclic so our simplified model is similar to hash-based directed acyclic graphs (HDAGs) described in [15].

In our model, each file system version is represented by one tree of blocks, with multiple trees sharing deduplicated blocks. Exposing pointers as part of block data not only makes block references explicit which simplifies deletion, but also allows for deduplication of blocks with pointers. This is very important for ensuring high dedup efficiency when subsequent backups do not differ much and backup differentials contain mostly blocks with pointers. An example of consecutive backups is presented in Figure 1(b).

Additionally, there is a mechanism to specify which of the trees are alive and should be preserved and which are dead and should be deleted. The details of this mechanism are described later.

3.2 Root set determination

In a traditional garbage collection, object accessibility is determined with respect to the *root set* usually composed of global and active local variables of a running program. Alive objects belong directly to the root set or can be reached from it by graph traversal. Since garbage collection in program runtime has full knowledge of program variables, computation of the root set is easy. However, it is more difficult in a storage system like described above, because its root set contains not only alive tree roots (corresponding to global variables), but also blocks which addresses can still be used by clients. These addresses are like values of local variables in programming languages, but the selection of such addresses by clients can be potentially arbitrary.

Without any restrictions on remembering block addresses by clients, potentially all blocks would constitute a root set. Therefore, we need to limit the set of addresses a client can keep in the least restrictive way for a client.

3.3 Supporting deduplication while running deletion

To avoid increased storage consumption caused by deletion, we reject solutions disabling deduplication during deletion. While it is running, an address of a block already scheduled for deletion can be returned to the client again as a result of deduplication. Since such block address can be used by the client, this block should not be removed even if deletion algorithm was about to decide otherwise. Therefore, a block resurrection through duplicate elimination must be traced by the deletion algorithm. For comparison, in the standard garbage collection, an inaccessible object cannot be resurrected. Preservation of deduplicated blocks is in fact a fundamental challenge for our deletion algorithm.

An alternative solution of writing new copies of duplicates of blocks to be deleted is not implementable, since we do not know which blocks will be marked for deletion until new counters are computed.

4 Client-visible semantics

In this section we extend the simplified model described in 3.1 to make it more realistic and to facilitate deletion. Besides enabling deletion, this model has been also validated with an implementation of a file system [30].

4.1 Block types

To access a tree of blocks, a client would have to remember cryptic block address of a root block. To avoid that, we introduce two block types: *regular blocks* and *named blocks*. Regular blocks are like described earlier and can be pointed by other blocks. Named blocks can be read with associated client-specified search keys. The search key can be an arbitrary string and is provided by the client along with binary data and pointers on named block write. For simplicity and to express intention of being root blocks, named blocks cannot be pointed by other blocks. Because named blocks are read with their keys only, the system prevents writing two named blocks with the same key and different content. Deduplication also works for named blocks.

The address of a regular block is not just a hash of its content; instead such address includes system-specific information unique to write which created this block. As a result, the block address cannot be derived from a block content by clients without writing such block first; therefore, block graphs can be built bottom-up only.

4.2 Deletion granularity

Removal of an arbitrary block should not be supported in a CAS-based system, as it could lead to creation of a dangling reference by deleting a block pointed by another existing block in the system. Instead, a CAS-based system should allow to mark all blocks reachable from a given named block as no longer needed.

To express this, we introduce two sub-types of named blocks (after [10]): *retention roots* and *deletion roots*.

A retention root marks blocks reachable from it as the ones to be preserved, whereas a deletion root ”cancels out” this property for the matching retention root. A deletion root matches to a corresponding retention root only if both have the same search key (this is an exception to the rule that any two named blocks should have their search keys different). Instead of having two different types of named blocks there could be just one type with retention flag in mutable block metadata. However, with a distinct deletion root there is a separate persistent request to remove a given tree of blocks. This allows

for simplification of the deletion algorithm, as it does not have to process deletion requests immediately, and instead can process them in batches as described later.

An example of graph of blocks can be found in Figure 2.

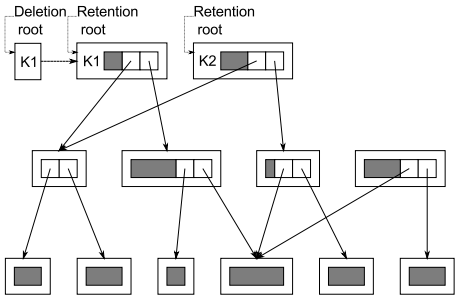

Figure 2: User-visible data model with named block types marked.

A retention root is *alive* if there is no matching deletion root in the system. Otherwise it is *dead*. Note that regular blocks not reachable from alive retention roots are treated as not to be preserved. An opposite approach would lead to leaving inaccessible garbage forever if a client application crashes during building a tree of blocks.

As discussed in 3.2, the root set includes also blocks remembered by clients, which leads to the following:

Invariant 4.1. (Invariant of block preservation) *The system preserves all blocks reachable from alive retention roots and blocks whose addresses can be used by clients.*

We call all blocks to be preserved as alive. If a block is not alive, we call it *dead*.

As explained earlier, to enable deletion we need to have an agreement between clients and the system that restricts the set of addresses which can be used by clients. Details of this protocol are given in the next section.

4.3 Restrictions on client-kept addresses

Since a client can potentially keep all obtained block addresses, we define restrictions on how clients can use them. This is done by introducing a virtual client-visible time built with so-called epochs. An *epoch* is a global counter with an *advance* operation. At any given point of time, the system is in only one epoch called the current epoch. We later show that frequency of advances is bound to frequency of garbage collection cycles.

Each block address obtained by a client has an epoch associated with this address. The system verifies on writing that an address with the associated epoch T can only be used in epoch T and $T + 1$. For reading, client can still use an address with an older epoch associated but the read may fail if the block was deleted.

An epoch associated with an address of a given block depends on how this address was obtained. The rules of epoch assignment are given in Table 1 and are also discussed below. These rules have been selected to ensure correctness of the deletion implementation, as shown in Section 5.6.

	Epoch of written block (new and deduplicated)	**Epoch of addresses within read block**
Regular block without pointers	current epoch	N/A
Regular block with pointers	minimum of epochs of block addresses inserted	epoch of supplied address
Retention root	N/A	current epoch

Table 1: Rules of epoch assignment for read and write operations.

On writing of a regular block without pointers the system returns a block address with the current epoch associated. When a block being written contains pointers, the epoch returned with this write is the minimum of epochs of these pointers (so-called *minimum rule*). Named blocks do not have regular addresses, so no epoch is assigned. An epoch returned on write does not depend on the duplicate status of a block being written.

Block addresses can also be obtained by reading blocks with pointers. Epochs of pointers obtained by reading a block are identical. For a regular block with pointers, this epoch is equal to the epoch associated with the address of the read block. For a named block with pointers, this epoch is equal to the current epoch.

Recall that the process of tree building requires a client to keep the addresses of already written blocks to insert them into parent blocks. However, after epoch introduction, only addresses stamped with the current and the previous epoch are allowed in successive blocks creation.

After the current epoch is advanced from T to $T + 1$, all clients are notified in order to discard all addresses with epoch T. Here discarding pointers means committing them to a tree pointed by an alive retention root. This needs to be completed before the epoch can be advanced to $T + 2$. In epoch $T + 2$, the system rejects writing blocks with pointers when even a single pointer has an address with an associated epoch smaller than $T+1$. The system also rejects attempts to read dead retention roots to restrict the set of block addresses that can be kept by clients. Such constraint is irrelevant for correctly behaving clients but essential for a correct implementation of

our deletion procedure.

Note that epochs of addresses are not persistent and exist only in memory of the system and its clients. In particular, disk representation of block with pointers does not contain any epochs, instead they are assigned to addresses only in the context of client operations according to the rules defined above.

With epochs, we can more precisely describe the invariant for block preservation because only blocks with addresses in the current and previous epochs can be used by clients.

4.4 Alternatives

It is possible to hide epochs from user by refreshing them in some intermediate layer which is informed by a client about addresses it wants to keep alive even though they are not yet reachable from any alive retention root. In such approach, this layer would automatically refresh such addresses on epoch changes as described above. Since there are only few client types so far, packaging epoch operations in such layer has not been done yet.

One more alternative is to provide an atomic operation to write a set of blocks with a temporary retention root (to be deleted later). This approach has a nice property that all blocks kept by the system are reachable from some retention root. However, a buggy or malicious client would be able to use addresses of deleted blocks and in consequence write a block with dangling pointers, which is impossible with epochs.

5 Deletion algorithm on block level

Below we describe the deletion procedure assuming it works on block level in a centralized system. The issues related to distributed storage architecture are addressed in later sections.

5.1 Deletion organization

Our deletion algorithm uses a well-understood technique of reference counting. Each block has a reference counter tracking the number of other blocks pointing to the counter owner. However, when a new block is written, counters of pointed blocks are not altered on-the-fly. Such immediate update would be very expensive due to possibly random I/O for every child block that can be located anywhere in the system. Instead, reference counters are updated in the background.

The whole deletion consists of two phases run periodically: *garbage identification* in which new counter values are computed and *space reclamation* during which storage space is reclaimed by removing garbage identified in the first part. As pointed out in Section 2, immediate reclamation is not necessary in secondary storage systems. Moreover, it is not advisable because of potential for increased resource consumption and required synchronization with ongoing deduplication.

In the remainder of this paper, by deletion we denote the garbage identification phase only. Deletion is started on demand, and in real usage scenarios its frequency varies from 1 per day to 1 per week. This frequency depends on available space (the less space, the higher frequency) and a particular backup policy implemented (for example it makes sense to run deletion each time the oldest backup is removed).

5.2 Base garbage identification

We now describe a base algorithm for garbage identification that serves as an extendable base for the final version of block-level deletion. The procedure as sketched here works smoothly but only if there are no writes overlapping with deletion. Handling of such case requires epoch mechanism and is described later.

Each run of garbage identification phase calculates counters of blocks written only up to a certain moment in time. For every block, there are two versions of counter during deletion: an effective one that is persistent and a temporary one used to store partial result of the deletion algorithm. On a deletion abort caused for example by a failure temporary counters can be discarded with no harm to the persistent ones.

Every started run divides blocks in three disjoint classes as shown in Figure 3. Membership in those classes is based on the time when block is written to the system for the first time. The first class contains blocks written before the last successful deletion start. We call them *done blocks*. The second class contains blocks written later but still before the current deletion start - we call them *todo blocks*. Finally, blocks written after the current deletion start belong to the third class called *new blocks*. Todo blocks will be processed by the current deletion to reflect not yet processed pointed-to relations to both todo and done blocks. This can result both in incrementation and decrementation of a counter of some done or todo block. New blocks are not processed in this deletion run and they do not influence counter values of other blocks for now. This influence will be computed in the next deletion run. In particular, all new deletion roots have no effect in this deletion run. Until the current deletion finishes, all new blocks are simply preserved and their counters are set to a special *initial value*. The next deletion run will process only new blocks which will become todo blocks for this run, i.e. the counters update process is incremental.

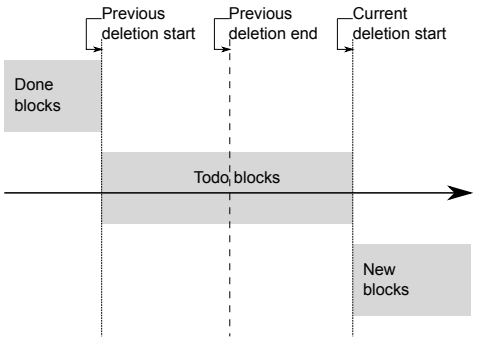

Figure 3: The statuses of blocks during garbage identification.

Each deletion run proceeds in subphases as follows. First, in the *incrementation subphase* all todo blocks that point to other blocks are read in and their target blocks counters are incremented. During such incrementation the initial value is treated as 0.

Next, in the *decrementation subphase* we first identify starting blocks for the counters decrementation. These are all retention roots with matching deletion roots as well as regular todo blocks that did not get their counters incremented (for now, we assume there are no new blocks pointing to them). Counters of such blocks are changed to 0.

All pointers from so identified garbage cause counters decrementation in blocks pointed by the garbage. Decrementation may also cause some other block counters fall to 0. Thus, new garbage is possibly identified again, and decrementation is repeated until no new garbage is found.

After that, in the *commit subphase* garbage blocks are made as no longer readable by making temporary counters persistent.

5.3 Space reclamation

We assume here a container-based storage organization similar to the one described in [10], but in a centralized system. A container keeps a set of blocks with associated block metadata. The system tries to keep a size of each container within a specified range of allowed sizes. Initially, a new container keeps all non-duplicate blocks written sequentially in a given window of time. After a new container reaches its desired size, it is closed, and newly written blocks are stored in a new container.

After garbage identification, space is reclaimed by a container sweep which discards deleted blocks. To speed up reclamation, priority is given to containers that have the highest ratio of data to be removed. After space reclamation, adjacent containers can be merged into one to avoid too small containers. Storage is reclaimed in the background as needed or when there are free cycles to perform the reclamation. The priority of reclamation increases when the system is close to become full.

5.4 Problems with the base garbage identification algorithm

Consider the case in Figure 4(a), in which a block A has been written just before the deletion start and a retention root R pointing to A has been written just after this point in time. Here, A is definitely not pointed by any other block but R which is new; hence, skipped by the identification process. As a result, A is removed mistakenly.

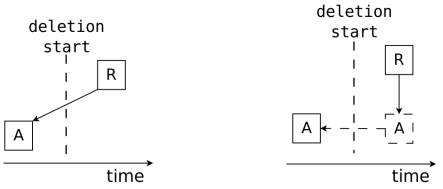

Figure 4: (a) Retention root is written to A after the deletion starts. A should not be deleted. (b) A becomes duplicate after deletion starts and should not be deleted.

The next problem is caused by deduplication and is illustrated by Figure 4(b). Consider a todo block A which is not pointed by any other block when the current deletion starts. If a client writes a duplicate block A and then a retention root R that points to A, both should be preserved. However, the deletion process will identify A as garbage since neither any todo nor old block points to it. Removing of A is then incorrect and leads to data loss.

5.5 Refinements of the base algorithm to accommodate new writes

5.5.1 Double advance

A solution to the problem illustrated in Figure 4(a) makes use of epoch restriction described in Section 4.3. Recall that a client is not allowed to keep an address older than the previous epoch. With deletion start there are two epoch advances, one after the other. As a result, no new references to block A can be created using an address of A stamped with old epoch after such double advance. Still, such reference can be created between the first and the second advance. To solve problem completely, we need to increment counters of blocks pointed by all blocks written for the first time between the first and the second advance. Note that we do not decrement counters of blocks pointed by such pointers, as it would lead to removal of some new blocks and violation

of our basic assumption to preserve unconditionally new blocks during deletion. Therefore, increments are always shifted by one epoch ahead as shown in Figure 5. Also, we should note here that references to A can also be created via deduplication during deletion but handling this cases is described in the next section.

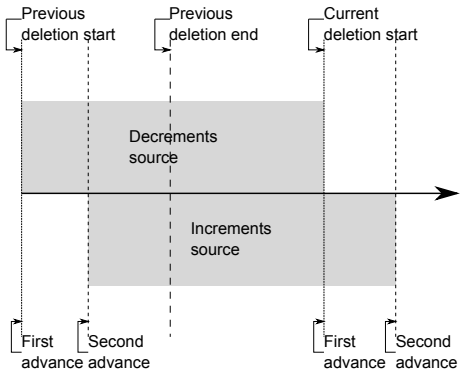

Figure 5: Borders of incrementation and decrementation during garbage identification when writes are allowed.

A deletion run covers one full epoch and a part of the next one as shown in Figure 5. The first epoch called *preparation epoch* allows clients to discard addresses with old epochs, as described above. In the second epoch, called *deletion marking epoch*, actual deletion computation is performed. There is no reason to bump current epoch at the end of the deletion run. Advance operations are coordinated by a separate management entity in the system that makes sure that the next advance is executed only if all clients confirmed the disposal of the old addresses. Misbehaving clients that do not confirm this within a limited time (for example due to a failure) are ignored.

The frequency of garbage identification automatically impacts the number of epoch advances. In theory, the frequency of epoch advances could be higher than the frequency of running the identification phase but there is no good reason to do so.

5.5.2 Preserving deduplicated blocks

To solve the problem shown in Figure 4(b), base blocks of duplicate writes are marked on write as non-removable during the current deletion by setting a so-called *undelete marker*. These markers are set from the beginning of deletion until start of the commit subphase. In this subphase, whenever a block is about to be logically removed (i.e. its newly computed counter is 0) and has the undelete marker set, such block is preserved by restoring its counter to the special initial value. As a result of this process called *undeletion*, the affected blocks are *undeleted*. Note that we do not undelete base blocks that are also new blocks because such blocks are not removed by the current deletion run anyway.

While the commit subphase is in progress we still need to deduplicate writes. However, continuing writing undelete markers in the commit subphase would create a race between setting and using them to restore counters to the initial value. Instead, we use the *base block filtering*. A write is deduplicated only in one of the following cases: the newly computed counter of the base block is positive; or undelete marker is set; or the base block is a new block. Otherwise, a fresh block is written even if its potential base block already exists in the system. This process allows for accurate deduplication during the commit subphase, i.e. it avoids resolving a write as a duplicate if the corresponding base block is just about to be deleted. Undelete markers are cleared on deletion completion.

Fast access to undelete markers is critical for deduplication speed, so we keep markers in RAM as bitmaps.

A completed deletion run may result in undeleted blocks with counter values set to the initial value. The next deletion run needs to decide if these blocks really should be kept. This is done with additional incrementation which is recursive, unlike regular incrementation from todo blocks. In a recursive incrementation, for each undeleted block from the previous deletion pointed by at least one of the new deletion todo blocks, we increment counters of pointed blocks and continue the increments as long as undeleted blocks are encountered. Counters of undeleted blocks not reached by recursive incrementation are simply changed to zero.

Note that, in the context of the last completed deletion run, undeleted blocks can only be pointed by other undeleted blocks or new blocks. If a given undeleted block is reachable from any new block, such undeleted block will be reached by recursive incrementation process described above. Otherwise, an unreachable undeleted block will be identified as garbage by this deletion run.

5.6 Deletion correctness analysis

We claim that our deletion procedure is correct, i.e. it does not remove blocks reachable from root set defined in Section 3.2. Ideally, we should also be able to show that garbage identification is complete i.e. removes all blocks not reachable from the root set. However, it is not the case, because all blocks written during the current deletion run are preserved: non-duplicates are preserved unconditionally, and duplicates are preserved because of undelete markers. In particular, such recently written blocks not reachable from any retention root are also preserved. This is only a minor problem, since we show below that such blocks will be deleted by the next

deletion run.

Now we introduce some terminology useful to prove our claims. All client operations that can affect the set of remembered addresses are writes, reads and discarding some of these addresses. The number of such operations executed during garbage identification is finite (n) and they are done in some order; let us assign consecutive natural numbers to them ($i = 1..n$). We denote as the i-th moment the time when the operation i finishes. Note that the first advance precedes any operation in this sequence and the second advance happens somewhere in the middle of it. Having the order defined, let $\{U_i\}_{i\in 1..n}$ be a sequence of sets of addresses remembered by a client restricted to addresses valid for writing, i.e. we assume that in epoch T the client does not keep addresses from epochs $T-2$ and earlier. Additionally, we define a parallel sequence $\{R_i\}_{i\in 1..n}$ of sets consisting of retention roots alive before the first advance or written after first advance. Sets R_i change with each newly written root after the first advance. Deletion roots written after the first advance are not taken into account here as they do not influence this deletion run. Let $S_i = U_i \cup R_i$. Operations that modify U_i are regular block writes, reads of any type of block with pointers, and discarding of kept addresses. Operations that modify R_i are retention root writes. Every operation changes only one of these sets.

Note that at i-th moment any member of U_i is readable. Also, note that at every point of time S_i is a superset of a root set, because S_i contains also retention roots which have already a deletion root associated written during this deletion run. To prove the correctness of the deletion procedure, it is enough to show that all blocks reachable from S_n are preserved. Using mathematical induction the stronger thesis could be proven.

Theorem 5.1. (Correctness property) *The following members of set S_i will be preserved by the deletion run that ends in epoch T:*

1. *all retention roots that belong to R_i,*
2. *all addresses in U_i with an associated epoch equal to $T-1$ or T,*
3. *addresses in U_i with epoch $T-2$ if pointed by any retention root from R_n,*
4. *all blocks reachable from (1), (2), or (3)*

The complete proof has been omitted due to space constraints. For almost all cases which modify set U_i an inductive step is trivial, non-trivial cases, like writing duplicates or reading old blocks, are covered by: undelete markers, base block filtering (see Section 5.5.2) and the fact that deletion preserves blocks with addresses with associated epoch $T-2$ reachable from retention roots written in system epoch $T-1$ (see Section 5.4).

Now we will define the class of blocks that the deletion removes.

Theorem 5.2. (Non-emptiness property) *A deletion run identifies as garbage the blocks that are written before the deletion starts and are simultaneously*

1. *not reachable from any retention root alive when the deletion starts and*
2. *not reachable from any retention roots written after the first and before the second advance and*
3. *not used as base blocks for duplicate elimination during this run.*

Proof. From setup of borders of increments and decrements, such blocks will have their counters computed as 0. If such blocks were not used as base blocks for duplicate elimination during deletion, these blocks are removed. □

As discussed above, blocks written during the current deletion run are preserved even when they are not reachable from any block in the root set. Applying non-emptiness property to the next deletion run shows that such blocks will be deleted in this run unless they are written again.

6 Deletion in distributed architecture

6.1 CAS-based distributed system

We assume a CAS-based distributed system similar to one described in [10] consisting of a set of dedicated storage nodes (SNs). Logically, the system is built around a distributed hash table (DHT) composed of *supernodes*. Each supernode is identified by prefix of hash space it is responsible for. Hash spaces of supernodes are disjoint and cover the entire hash space. Every supernode is responsible for handling client writes and reads of blocks with content-based hashes belonging to the hash space of this supernode. The supernode itself is divided into a fixed number of *peers* usually distributed over SNs to provide node failure resiliency. Figure 6 presents exemplary distribution of peers over physical machines in the system with 4 supernodes and number of peers within a single supernode equal to 4.

On write, a block is routed to appropriate supernode, erasure coded and the obtained *fragments* are then distributed to peers that append received fragments to fragment containers which are separate for each peer. Figure 7 visualizes stream chunked into blocks, and focuses on block A that is routed to supernode 01, fragmented into fragments A_i sent to peers and appended there to fragments containers.

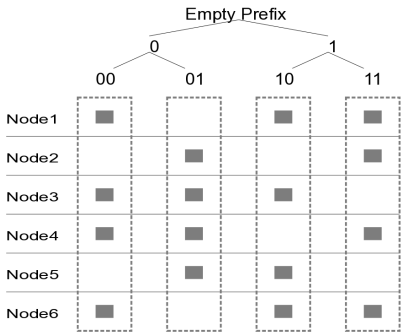

Figure 6: Supernodes, peers

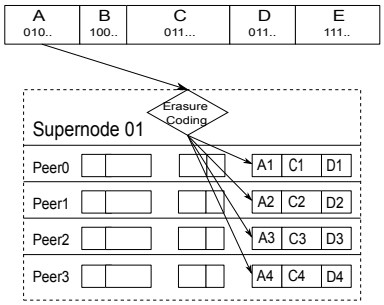

Figure 7: Fragments, fragment containers

The architecture described above delivers dynamic and failure-tolerant system. In particular, peers are not statically bound to storage nodes. Instead, once a node becomes unavailable, peers are reconstructed on different nodes along with peer data. Similarly, when a new node is added, some peers are transferred to it. Therefore, some data may not be available in the current location of a peer until this data is transferred or reconstructed.

6.2 Challenges for deletion introduced by distributed architecture

Distributed architecture complicates application of deletion algorithm on block level presented earlier. A deletion algorithm must be able to locate necessary data and metadata in spite of their changing locations. The algorithm must also make a consistent decision to preserve or remove all fragments of one block. Beyond fail-stop failures, deletion should cope with nodes that are temporarily unavailable due to intermittent network failures and which will re-appear sometime in the future. In particular, a deletion algorithm should be able to distinguish fragments of logically removed blocks found on previously dead machines from the fragments of blocks that are still alive.

6.2.1 Redundancy of computation

To provide failure tolerance, for each block we replicate its persistent counter with all block fragments. Additionally, temporary counters and undelete markers are computed independently by multiple selected peers in each supernode. These peers are called *good peers* and must have good-enough data state: for example, all blocks with pointers must be present in the current peer location.

Since each good peer performs identical tasks to identify garbage blocks, there is a high redundancy in deletion computation. Deletion will not start until sufficient number of good peers can be identified in each supernode. The remaining peers are idle during the deletion algorithm and eventually receive final counters from others in the background after the deletion ends. Any peer failure reduces the set of good peers. If the number of good peers within any supernode drops below some configurable minimum, the whole deletion process is aborted and its partial results are abandoned.

Undelete marker setting and base block filtering are done individually on each peer. Setting an undelete marker for a given base block needs to complete on all good peers before a write is resolved as a duplicate. Similarly, the result of base block filtering is computed as a logical conjunction of answers from all good peers.

Undeletion in the distributed architecture is done independently on each good peer, so it is critical to have all markers consistent across good peers. Undelete markers on each peer can be inconsistent because undelete marker writes issued earlier may fail leaving markers set only on some good peers. Markers are made consistent by applying logical conjunction of marker bits from all good peers within a supernode and distributing the result back to all good peers. After making the markers consistent each peer applies the markers to temporary counters kept by this peer.

To recognize obsolete counters, each persistent counter is stamped with deletion marking epoch of the latest deletion run in which this counter value was valid. For all fragments, counter stamps are refreshed on each good peer by setting stamp value to the deletion marking epoch. Unreachable peers get their counters invalidated implicitly, because stamps associated with fragments kept by such peers are not refreshed.

6.2.2 New counters commit

Once the undeletion is completed, the equality of computed counters is verified for each supernode. Counters mismatch on any supernode would indicate software failure. If any mismatch is found, the deletion process is aborted. After successful verification, temporary counters are made persistent and the system is ready to use

them. Good peers vote to enter proper commit subphase in which the system starts to use new counters. From now on, each node restarted after failure strives to complete counters commit. The switch itself is quick as it requires only the replacement of metadata of all fragment containers.

7 Deletion evaluation

The deletion algorithm satisfies all requirements mentioned in Section 2. The algorithm observes a well-defined block aliveness semantics introduced earlier. All alive blocks are preserved, most dead blocks are deleted. Some potential garbage blocks may be temporarily kept by the system because the deletion algorithm is conservative. However, such blocks are kept only until the next deletion run, which will identify these blocks as garbage.

Clients are able to write concurrently to deletion because of the epochs advance restriction described in Section 4.3. The lightweight mechanism of undelete markers allows for good deduplication effectiveness and write performance while the deletion is running.

As the performance evaluation presented in Section 8 shows, the deletion causes quite limited drop in overall user-visible system performance because of design decisions like: (1) deferring space reclamation and reference counters update; (2) efficient distribution of increments and decrements and their application in batches to temporary counters; (3) starting space reclamation from containers with the highest ratio of space to be reclaimed; and (4) keeping undelete markers in good peers memory.

The scalability of the deletion algorithm is similar to the scalability of the entire system as deletion computation is distributed among supernodes. The deletion copes with the dynamic nature of the system using internal DHT in the same way as it is done for data location for user writes and reads.

The deletion process is resilient to peer failures. They rarely terminate the garbage identification process due to redundancy of computation among peers. The identification continues as long as there is sufficient number of good peers for each supernode. Counters verification, described in Section 6.2.2, ensures the consistency of the decision to keep or remove blocks on good peers and gives protection from the most harmful software failures. Additionally, counter stamps allow for identification of obsolete metadata.

8 Performance evaluation

Our deletion algorithm has been implemented in a commercial system called HYDRAstor designed as a storage target for backup and archival data. It is a distributed system - besides storage nodes (SNs), the system consists of a front-end built of so-called access nodes (ANs). A typical HYDRAstor deployment varies from 1 to 8 SNs, but there are also larger installations consisting of tens of SNs. The system uses content-derived block addresses and provides global in-line block deduplication. HYDRAstor distributed architecture is similar to the one described earlier in 3.1. The system supports self-recovery from failures and uses erasure codes to provide multiple user-selectable data resiliency levels.

All experiments except where noted use the 3rd generation HYDRAstor composed of 4 storage nodes and 2 access nodes. All servers run the Red Hat EL 5.4 Linux, have 2 quad core 2.4GHz CPUs and 4 GigE cards. Each SN has 24GB of RAM and 12 1TB SATA disks, whereas each AN has 12GB of RAM and a limited local storage. The system is configured with 8 supernodes, each consisting of 12 peers.

The experiments were performed with the average block size of 64KB compressible by 33%. The data blocks were written as 9 original and 3 redundant fragments providing resiliency to 3 disk failures. We measured client-side bandwidth only, without contribution of redundant fragments.

HYDRAstor allows user to alter default resource division by setting the division of system resources among user load, deletion and other background tasks. For instance, a user can assign only minimal resources to deletion to have the lowest impact of deletion on user load which results in slower garbage identification and reclamation. Alternatively, a user can trade certain user load performance degradation to speed up deletion. By default, the percentage of resources assigned to deletion is 30%.

8.1 Garbage identification vs. user reads and writes

This experiment measures the mutual impact of garbage identification and user operations like reads and writes.

Each measurement started with a prologue simulating client backup activity with the following 5 steps: filling an empty system with $E = 16$ TB of non duplicated data; running an initial deletion to compute block counters; loading $C = 1.6$ TB of new data; writing C TB of existing data and writing deletion roots marking for deletion C TB of data.

We measured the impact of garbage identification on performance of user writes as a function of dedup ratio of the data written. We collected 15 data points in total, in groups of 3 for each of the 5 values of dedup ratio varying from 0% up to almost 100% with a step of 25%. In each group, the first number indicates write bandwidth when no deletion is running; the second with

deletion running and assigned default maximum 30% of resources; and the third also with deletion running, but with only minimal (maximum 1%) resources assigned to it. We recorded duration of garbage identification when writing and also with no user load present.

The results are shown in Figures 8 and 9. When deletion is given default 30% of system resources, user writes are slowed down by less than this fraction. Moreover, in such default setting, deletion is prolonged by less than three times compared to the case when no user load is present. (see Figure 9). When deletion is configured to use minimal resources, its impact on user writes is below 5%. In general, handling of undelete markers affects end performance of writes just negligibly although in fact we observed increased CPU consumption on storage nodes.

The impact of the deletion on user reads is smaller than on writes, with only 10% reduction of read bandwidth (1024 MB/s vs 1135 MB/s) when deletion is given up to 30% of resources. In such case, deletion is prolonged by a factor of 2.27 as a result of user reads. This impact is practically negligible when deletion runs with only minimal resources assigned. This is because reads do not require additional action during deletion, unlike in the case of duplicate writes.

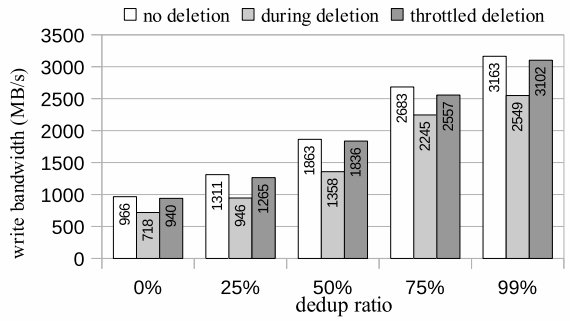

Figure 8: Write bandwidth during garbage identification. Throttled deletion is given only 1% of resources instead of standard 30%.

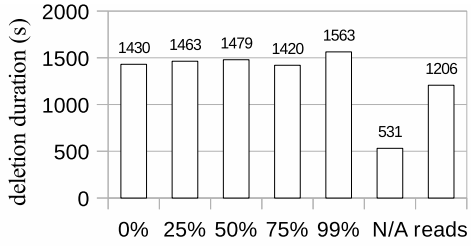

Figure 9: Duration of garbage identification with 30% resource share while user operations (writes or reads) are in progress. Percentages on x-axis denote fraction of duplicate writes. N/A stands for no user load.

8.2 Deletion after node failure and all data deletion

Using the same prologue as in 8.1 we measured duration of garbage identification with no user load present after failure of one storage node. It took only 20% more compared to the case of all nodes healthy. Finally, garbage identification after removal of 16 TB from the healthy system after completion of the prologue took 1066 seconds which translates into 15.4 GB/s of logical deletion speed.

8.3 Deletion scalability

The goal of this series of experiments is to examine how deletion performance changes when the system grows. We used the 2nd generation storage nodes which have slightly slower disks and CPU than the 3rd generation nodes. In four individual experiments number of storage nodes n is varied from 4 to 16 with a step of 4. For each point the prologue defined in 8.1 is performed, but with data amount now depending on number of storage nodes: $E = n{*}38$ TB and $C = n{*}3.8$ TB. We measured duration of initial deletion (i.e. computing the counters, which is the second step of the prologue) as well as duration of garbage identification with no user load present (this step identifies data marked for deletion in the last step of the prologue). Results in Figure 10 show that duration of garbage identification remains roughly the same when the system grows in number of storage nodes and the size of data used per storage node remains constant.

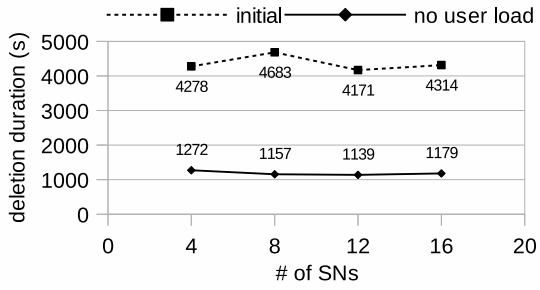

Figure 10: Duration of garbage identification when the number of storage nodes grows.

9 Related work

The implemented deletion bears some resemblance to concurrent garbage collection in programming languages with dynamic data structures which is well-understood [2, 32]. However, there are significant differences. On the one hand, our deletion is more complicated because of deduplication, scalability and failure tolerance of CAS-based distributed storage system;

on the other hand, the deletion is simplified because the graphs of blocks must be acyclic which is not the case in programming language-related garbage collection.

Our deletion is based on deferred reference counting [32]. Alternatively, we could have used mark and sweep [8] to implement our deletion. We decided against it to avoid traversing all data on each deletion run.

Grouped mark and sweep (GMS) [11, 18] attempts to solve this problem by grouping backups and marking only those containers which keep objects from groups with some files added or deleted since the previous run of the algorithm. With grouped mark and sweep, old mark results are preserved for each group and each container. With approximately constant size of each group, scalable storage and high dedup ratio such results may be non-empty, which will result in large metadata per container.

Our undelete markers are in a way similar to graying of a white object on read in the tricolor mark-and-sweep scheme [32]. GMS authors propose also to handle deduplication during deletion with a special in-memory protection map. Such map keeps fingerprints of blocks used for deduplication during deletion and prevents such blocks from being reclaimed. However, if deletion runs for too long, the protection map may not fit in the memory, whereas undelete markers by design avoid this problem.

Our epochs may superficially resemble generations in garbage collection theory [32]: new blocks are written to the new generation whereas every deletion run moves surviving blocks to the old generation. However, our algorithm is not generational at all as blocks from old generations not reachable from alive retention roots are garbage-collected during every deletion run.

Without deduplication, deletion in a distributed storage system is relatively simple and can be done with leases like in Glacier [19], or with simple reclamation of obsolete versions like in Ursa Minor [1]. However, with deduplication, deletion becomes difficult for reasons explained earlier. For example, Venti [27], Deep Store [33] and Sparse Indexing [22] have not implemented deletion. Another class of systems implements garbage collection on their storage units in a disruptive manner. MAD2 [31] reference-counts fingerprints but freezes all involved tankers during the physical deletion period. DDE [20] revokes all data locks held by clients to free dereferenced blocks. dedupv1 [23, 24] in background marks unreferenced blocks as ”garbage collecting candidates” but commits their removal only if the system is idle. Other works include: usage of backpointers (reference lists) in SIS [6] and in FARSITE [3]; collecting unused blocks during exclusive scanning of a part of global hash index updated out-of-band in DeDe [7].

Data Domain [25, 26, 34] patented a garbage collection procedure in a centralized system with inline deduplication. Selection of blocks for removal is done there using Bloom filter which results in some garbage remaining in the system. EMC Centera [13, 17, 29] patented an explicit deletion of a content unit but does not mention how concurrent deletion and deduplication is handled; Extreme Binning [5] localizes their deduplication within *bins* and claims this easies garbage collection although no details are given.

10 Conclusions and future work

We have described the deletion algorithm for a scalable distributed storage with deduplication. Our deletion has been implemented and deployed in a commercial system - HYDRAstor. The algorithm allows for deletion to proceed concurrently to user reads and writes. Moreover, it satisfies other critical functional requirements such as high availability, limited performance impact on user operations and resiliency to multiple disk, node and network failures.

Epoch mechanism and undelete markers are two key techniques allowing for writing with deduplication while deletion is running. By creating boundary between old and newly written data and limiting set of block addresses kept by clients, epochs define clear semantics for a concurrent deletion process. Together with undelete markers this semantics enables also deduplication of data scheduled for deletion.

Performance impact, in turn, is reasonable because deletion operations are performed in batches and this work is distributed over the entire network. Importantly, undelete markers are kept in the main memory which results in a low overhead of marker handling on user writes.

Failure tolerance is achieved by redundancy of computation associated with good peers. Selected peers perform critical computations redundantly allowing the deletion process to proceed even if several of them crash. Good peers are also responsible for deletion procedure scalability, which is ensured by distributing deletion work among supernodes, without having any centralized component.

Although deletion is fully functional today, important features could still improve its value for the end user. Since the deletion procedure is concurrent, most improvements involve ensuring further performance boosts. One of the potential improvements is an introduction of separate containers for blocks with pointers which may speed up the counter incrementation subphase. Apart from improving performance, other directions for future work include making the deletion restartable by checkpointing intermediate deletion results.

References

[1] ABD-EL-MALEK, M., II, W. V. C., CRANOR, C., GANGER, G. R., HENDRICKS, J., KLOSTERMAN, A. J., MESNIER, M. P., PRASAD, M., SALMON, B., SAMBASIVAN, R. R., SINNAMOHIDEEN, S., STRUNK, J. D., THERESKA, E., WACHS, M., AND WYLIE, J. J. Ursa Minor: Versatile Cluster-based Storage. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies* (2005).

[2] ABDULLAHI, S. E., AND RINGWOOD, G. A. Garbage Collecting the Internet: A Survey of Distributed Garbage Collection. *ACM Comput. Surv. 30*, 3 (1998), pp. 330–373.

[3] ADYA, A., BOLOSKY, W., CASTRO, M., CHAIKEN, R., CERMAK, G., DOUCEUR, J., HOWELL, J., LORCH, J., THEIMER, M., AND WATTENHOFER, R. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment, 2002.

[4] ANDROUTSELLIS-THEOTOKIS, S., AND SPINELLIS, D. A Survey of Peer-to-Peer Content Distribution Technologies. *ACM Comput. Surv. 36*, 4 (2004), pp. 335–371.

[5] BHAGWAT, D., ESHGHI, K., LONG, D. D. E., AND LILLIBRIDGE, M. Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup. In *Proceedings of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2009)* (Sep 2009).

[6] BOLOSKY, W. J., CORBIN, S., GOEBEL, D., AND DOUCEUR, J. R. Single Instance Storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium* (2000), pp. 13–24.

[7] CLEMENTS, A., AHMAD, I., VILAYANNUR, M., AND LI, J. Decentralized Deduplication in SAN Cluster File Systems. In *Proceedings of the USENIX Annual Technical Conference* (June 2009).

[8] DIJKSTRA, E. W., LAMPORT, L., MARTIN, A. J., SCHOLTEN, C. S., AND STEFFENS, E. F. M. On-the-Fly Garbage Collection: An Exercise in Cooperation. *Commun. ACM 21*, 11 (1978), 966–975.

[9] DONG, W., DOUGLIS, F., LI, K., PATTERSON, H., REDDY, S., AND SHILANE, P. Tradeoffs in Scalable Data Routing for Deduplication Clusters. In *Proceedings of the 9th USENIX conference on File and Storage Technologies* (Berkeley, CA, USA, 2011), FAST'11, USENIX Association, pp. 15–29.

[10] 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: Proceedings of the 7th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2009), USENIX Association, pp. 197–210.

[11] EFSTATHOPOULOS, P., AND GUO, F. Rethinking Deduplication Scalability. In *2nd USENIX Workshop on Hot Topics in Storage and File Systems* (Boston, MA, USA, June 2010).

[12] EMC Avamar: Backup and Recovery with Global Deduplication, 2008. http://www.emc.com/avamar.

[13] EMC Centera: Content Addressed Storage System, January 2008. http://www.emc.com/centera.

[14] EMC Corporation: Data Domain Global Deduplication Array, 2011. http://www.datadomain.com/products/global-deduplication-array.html.

[15] ESHGHI, K., LILLIBRIDGE, M., WILCOCK, L., BELROSE, G., AND HAWKES, R. Jumbo Store: Providing Efficient Incremental Upload and Versioning for a Utility Rendering Service. In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies* (Berkeley, CA, USA, 2007), USENIX Association, p. 22.

[16] Exagrid. http://www.exagrid.com.

[17] GUNAWI, H. S., AGRAWAL, N., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND SCHINDLER, J. Deconstructing Commodity Storage Clusters. In *Proceedings of the 32nd International Symposium on Computer Architecture* (Madison, WI, June 2005).

[18] GUO, F., AND EFSTATHOPOULOS, P. Building a High-performance Deduplication System. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference* (Berkeley, CA, USA, 2011), USENIX ATC'11, USENIX Association, p. 25.

[19] HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation* (Berkeley, CA, USA, 2005), USENIX Association, pp. 143–158.

[20] HONG, B., AND LONG, D. D. E. Duplicate Data Elimination in a SAN File System. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST* (2004), pp. 301–314.

[21] IBM. IBM ProtecTIER Deduplication Solution http://www-03.ibm.com/systems/storage/tape /protectier/

[22] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZIS, G., AND CAMBLE, P. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *FAST'09: Proceedings of the 7th USENIX Conference on File and Storage Technologies* (2009), pp. 111–123.

[23] MEISTER, D., AND BRINKMANN, A. dedupv1: Improving Deduplication Throughput Using Solid State Drives (SSD). In *Proceedings of the 26th IEEE Symposium on Massive Storage Systems and Technologies (MSST)* (May 2010).

[24] MEISTER, D., AND BRINKMANN, A. dedupv1: Improving Deduplication Throughput Using Solid State Drives (Technical Report). Tech. rep., April 2010.

[25] PATTERSON, R. H. Incremental Garbage Collection of Data in a Secondary Storage. Patent, 11 2008. US 7451168.

[26] PATTERSON, R. H. Probabilistic Summary Data Structure Based Encoding for Garbage Collection. Patent, 04 2010. US 7707166.

[27] 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.

[28] SEPATON Scalable Data Deduplication Solutions. http://sepaton.com/solutions/data-deduplication.

[29] TODD, S. J., KILIAN, M., TEUGELS, T., MARIVOET, K., AND MATTHYS, F. Methods and Apparatus for Managing Deletion of Data. Patent, 04 2010. US 7698516.

[30] UNGUREANU, C., ARANYA, A., GOKHALE, S., RAGO, S., ATKIN, B., BOHRA, A., DUBNICKI, C., AND CALKOWSKI, G. HydraFS: A High-throughput File System for the HYDRAstor Content-addressable Storage System. In *FAST '10* (2010), pp. 225–239.

[31] WEI, J., JIANG, H., ZHOU, K., AND FENG, D. Mad2: A Scalable High-throughput Exact Deduplication Approach for Network Backup Services. In *Proceedings of the 26th IEEE Symposium on Massive Storage Systems and Technologies (MSST)* (May 2010).

[32] WILSON, P. R. Uniprocessor Garbage Collection Techniques. In *IWMM '92: Proceedings of the International Workshop on Memory Management* (London, UK, 1992), Springer-Verlag, pp. 1–42.

[33] YOU, L. L., POLLACK, K. T., AND LONG, D. D. E. Deep Store: An Archival Storage System Architecture. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 804–815.

[34] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2008), USENIX Association, pp. 1–14.