Sketches of Space: Ownership Accounting for Shared Storage

Jake Wires, Pradeep Ganesan, and Andrew Warfield

Coho Data

jake,pradeep.ganesan,andy@cohodata.com

ABSTRACT

Efficient snapshots are an important feature of modern storage systems. However, the *implicit sharing* underlying most snapshot implementations makes it difficult to answer basic questions about the storage costs of individual snapshots. Traditional techniques for answering these questions incur significant performance penalties due to expensive metadata overheads. We present a novel probabilistic data structure, compatible with existing storage systems, that can provide approximate answers about snapshot costs with very low computational and storage overheads while achieving better than 95% accuracy for real-world data sets.

CCS CONCEPTS

• **Information systems → Thin provisioning**; *Version management*; • **Theory of computation → Sketching and sampling**;

KEYWORDS

Snapshots, Copy on Write, Implicit Sharing, Space Accounting

ACM Reference Format:

Jake Wires, Pradeep Ganesan, and Andrew Warfield. 2017. Sketches of Space: Ownership Accounting for Shared Storage. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 13 pages.

https://doi.org/10.1145/3127479.3132021

1 INTRODUCTION

The ability to quickly snapshot data is a foundational mechanism in building cloud infrastructure. Whether the underlying storage abstraction is an object, file, or volume,[1] snapshots allow for the fast provisioning of VMs against a "gold master" image [33], support fast consistent backups of file systems [20, 28], and provide a rollback-based recovery mechanism in support of administrative changes [5]. Moreover, where implementations allow for very fast checkpoint operations and very large numbers of persisted snapshots, they enable a fork/join model of computing that has applications in failure recovery [36], parallel computation [27], concolic testing [8], and outbreak detection [48].

Mechanistically, snapshots are just a layer of indirection: by virtualizing an object's address space, it becomes possible to quickly and efficiently create additional *copy on write* (CoW) versions of it. With CoW, only new, divergent, data is written to the new address space, resulting in large reductions in both capacity and provisioning time.

However, the power of snapshots comes with a subtle and somewhat painful complexity: *implicit sharing*. While it is reasonably easy to use virtualization to quickly snapshot one address space into many, the resulting multitude of references from virtual objects back onto shared data makes accounting for that data hard. Precise ownership and liveness of individual blocks of data within a snapshot tree are expensive to determine and add metadata complexity in the form of reference counts and backpointers [11, 25, 30, 38]. This metadata complexity is problematic both because it introduces overhead on the data path in the form of dependent updates and because it complicates failure recovery. Not surprisingly, it has been the source of a number of defects in publicly available storage systems [21, 35, 43, 45].

As a result, systems that support snapshots, especially at significant scale, frequently perform accounting-related operations such as capacity reporting, quota enforcement, and garbage collection as long-running background tasks. These tasks, in performing full scans of snapshot metadata, risk wasting time scanning low-value regions of a potentially large metadata tree and also result in stale accounting information being reported to administrators and quota enforcement policy [22, 29].

This tension between the performance overhead and complexity of precise ownership-related metadata on one hand and the lack of timeliness and wasted effort involved in background scanning on the other is a core design challenge in snapshot implementations. In this paper, we observe that it is also largely unnecessary. Complex metadata structures such as backpointers do facilitate accounting operations, but they are really just a proxy for a small group of capacity-oriented set operations between related snapshots. We demonstrate this by showing how tasks such as calculating an individual snapshot's size in the face of implicit sharing or identifying high-reward candidates for garbage collection can be sensibly articulated as union and intersection operations performed across nodes in a snapshot tree.

Borrowing from the application of compact data structures to estimate query sizes in database research, we observe an opportunity to embed an accurate, space-efficient estimate of both capacity

[1] we will use these terms interchangeably throughout the paper.

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA

ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3132021

and address space utilization within each snapshot. By selecting compact data structures that support set cardinality operations, we are able to implement a low-overhead and easily-maintained accounting layer that is independent of underlying address-level data structures. This decoupling of metadata provides a clean separation of concerns and allows space accounting to be computed efficiently and independently from the rest of the snapshot implementation.

The system we describe in the rest of this paper is based on a hybrid, load-responsive data structure that combines roaring bitmaps [4] to precisely account consumption for relatively small snapshots with probabilistic counters [3] to estimate consumption in large snapshots. This approach is practically deployable within existing snapshot implementations today. It provides an immediate solution for users and administrators who frequently request improved visibility into storage consumption [10, 37, 50, 51], and it can help garbage collection and quota subsystems operate more efficiently by enabling better scheduling and prioritization of per-snapshot analysis tasks.

2 BACKGROUND

Storage systems have supported snapshots in one form or another for decades, using a variety of different techniques. Snapshots were originally implemented for LUNs, volumes, and other block-level targets [13, 28], but they have now been integrated directly into a number of file systems that support snapshot operations on directories and even individual files [20, 39, 40, 42]. This functionality has been particularly important in the realm of hardware virtualization, where it plays a fundamental role in the creation and management of virtual disks [5, 33].

Some very early systems implemented snapshots directly on top of existing volume mirroring technology, a technique that was subsequently referred to as *split-mirror* [1]. By intentionally breaking replication at a given point in time, early implementations of these systems supported the creation of exactly one volume snapshot, albeit at the cost of very high capacity overhead. As the utility of snapshots became increasingly evident, more efficient techniques were developed.

2.1 Copy on Write

Modern systems support efficient snapshots by using address space virtualization to enforce *copy on write* or *redirect on write* semantics. Both of these schemes enforce special write policies that allow physical blocks to be safely shared among multiple snapshots. In the former approach, shared blocks are copied to new locations before updates are applied in place. In the latter approach, updates are applied to new physical locations and shared blocks remain unmodified. In either case, address space virtualization is leveraged to efficiently update address mappings when blocks are relocated. [2] As Table 1 shows, most snapshot implementations prevalent today

[2]Because we are here concerned with the *ownership* of blocks, but not their physical locations, these two schemes can be treated identically for our purposes (§ 3). For clarity of exposition, we use the term 'copy on write' to refer to both.

Storage System	Split-Mirror	Overlay	Hierarchical
btrfs			✓
EMC Symmetrix	✓		
Hitachi ShadowImage	✓		
LVM		✓	
QCoW		✓	
QCoW2			✓
UFS		✓	
Veritas FlashSnap	✓		
VMDK		✓	
VHD		✓	
WAFL			✓
ZFS			✓

Table 1: CoW addressing schemes of popular storage systems

are variants of one of two addressing schemes: *overlay addressing* and *hierarchical addressing*.

2.1.1 Overlay Addressing. With overlay addressing [32, 46, 47] (also commonly referred to as *linked cloning*), an image is snapshotted by marking it read-only and linking a new sparse, empty image to it. The new image initially shares all data with the original base image; sharing is broken when new writes are applied to the linked image (Figure 1). Special metadata, typically stored in or alongside the image, records which addresses have been overwritten; this metadata usually takes the form of bitmaps or lookup tables, although alternative forms like undo logs also exist. Reads are directed to the linked image if the metadata indicates the requested address has been overwritten; otherwise, the read is forwarded to the previous image in the chain, where this process repeats.

Both read-only snapshots and writable images can be snapshotted by creating new, empty linked images. In this manner, a snapshot family can be seen as a tree with images as nodes and links as edges. Trees can scale to arbitrary breadth and depth as snapshots are created, although naive implementations exhibit high read latencies for large tree depths.

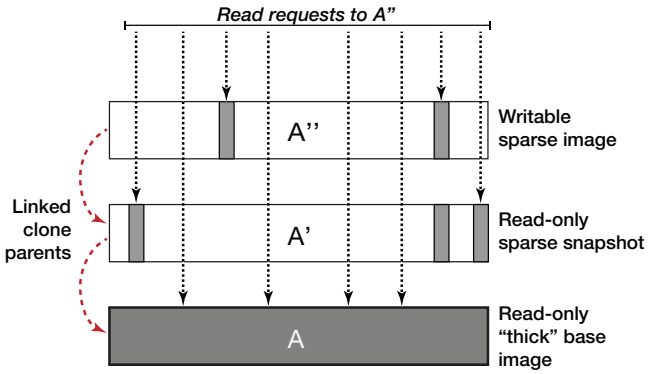


Figure 1: Copy on write via overlay addressing. Writes are applied directly to the top writable image (A''). Reads are passed from one image to another until the requested data is located.

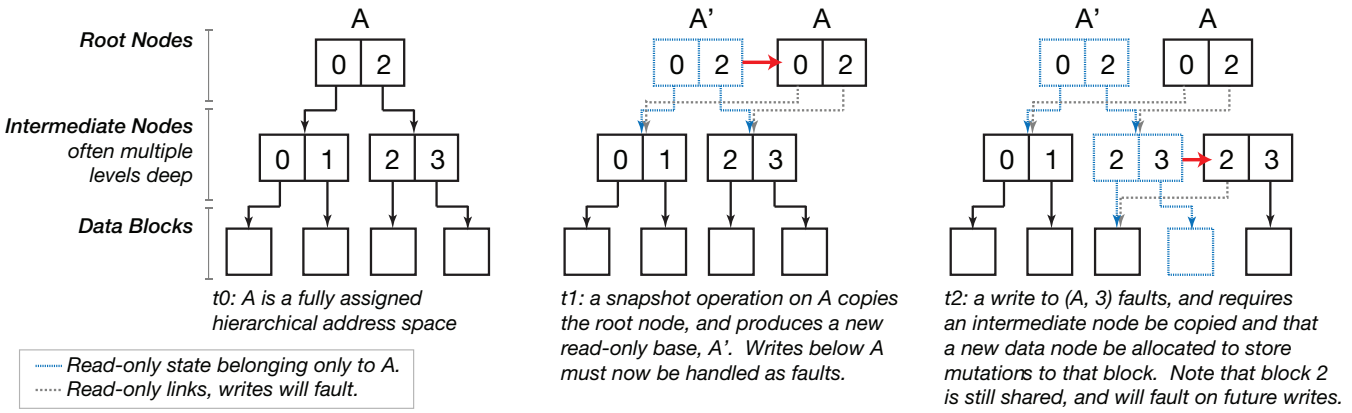


Figure 2: Preserving read performance at the expense of write complexity in hierarchical addressing.

2.1.2 Hierarchical Addressing. Hierarchical addressing [33, 39, 40] solves the problem of poor read performance for long overlay chains by maintaining explicit translation maps for snapshots and all derived images. This map is typically structured as some variant of a persistent tree [12] that associates virtual offsets in an image with the physical locations of the corresponding data, as depicted in Figure 2. An image is snapshotted by marking the tree and its referents read-only and creating a new writable copy of the tree's root. When shared data is to be overwritten, a new unit of storage is allocated for the incoming data, and the writable image's tree is updated to remap the virtual address to this new location. Updates to shared internal nodes in the tree must also be redirected to new storage locations, and this process may potentially bubble all the way to the root node. This approach may incur more overhead than overlay addressing when shared data is overwritten, but it ensures a constant overhead on the read path and thus generally scales better for deep snapshot trees.

2.2 Implicit Sharing

As a matter of efficiency, copy on write techniques rely on *implicit sharing*: shared data need not be explicitly linked to all the images that reference it. While this reduces the cost of creating new snapshots and overwriting shared data, it introduces new difficulties in tracking ownership over time. In particular, it is not trivial to determine which data is unique to a given snapshot and which data is shared by other images.

To see why this is so, consider the scenario depicted in Figure 3. In this figure (and throughout the rest of the paper), we model snapshot families as trees in which leaf nodes represent user-visible images or snapshots and internal nodes represent points in time at which snapshots were taken. Writes are only applied to leaf nodes. Leaf nodes (and only leaf nodes) can be snapshotted by linking two or more empty children to them, converting them to internal, read-only nodes in the process.

Notice how block ownership changes over time in Figure 3. At time t_0, B and C are empty, and they both share logical blocks $\{0, 1, 2\}$ with node A. At time t_1, block 2 in node C has been overwritten and C has been subsequently snapshotted to create new nodes D and E, both of which have been written to as well. At this point in time, node B has exclusive ownership of block 2 (because it has been overwritten in node C) as well as block 0 (because it has been overwritten in *both* D and E). At time t_2, when block 1 is overwritten in node E, node B takes exclusive ownership of *all* blocks from node A.

This example illustrates how implicit sharing complicates the notion of block ownership in snapshot families. In particular, the set of blocks exclusive to node B changes as a consequence of operations performed on other nodes in the tree, even though B itself has not been modified. This can cause difficulties when implementing features like garbage collection and quota enforcement, which benefit from being able to quickly calculate space consumption on a per-node basis.

There are two common approaches to solving this problem: the first involves periodically scanning metadata to compute the *reachability* of physical data, while the second involves maintaining *reference counts* to track this information explicitly.

2.2.1 Reachability. In their simplest forms, both overlay and hierarchical addressing maintain only forward references: address translation metadata makes it easy to map an offset within an image to its physical location, but no reverse mapping is maintained. In order to determine whether a given disk address is *live* (i.e., referenced by one or more images), it is necessary to scan the metadata of *all* images in the system [33, 41].

This is generally done using some variant of a *mark and sweep* [31] algorithm, which builds up an in-memory description of which data can be reached from live images; any data that cannot be reached from any live images may be assumed to be unused and eligible for reclamation. Scanning all image metadata is an expensive task, and as such this algorithm typically only runs periodically (say, a few times a day), or when disks are nearly full.

The classical mark and sweep algorithm does not determine how much storage an individual snapshot or image consumes. To answer this question, the algorithm must be modified to consider only disk

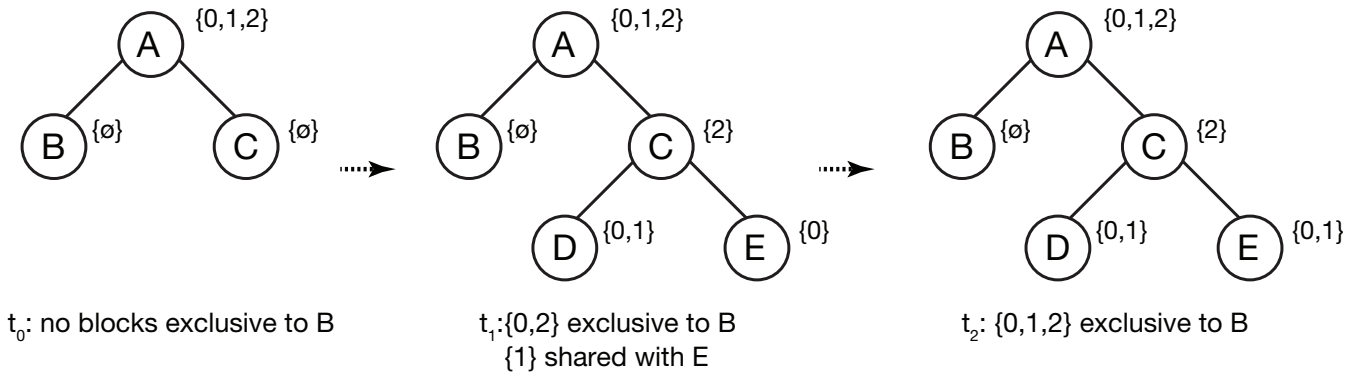


Figure 3: Evolution of implicit sharing over time in a snapshot tree.

addresses referenced by a single snapshot family, and to look for addresses only referenced by the snapshot in question (§ 3.2).

Techniques like mark and sweep that compute reachability asynchronously are convenient because they do not require additional persistent metadata. However, due to the high cost of these approaches, they are not well-suited for providing accurate, timely information about snapshot space usage, which can change rapidly in online systems. For example, the ownership information reported by garbage collection can grow stale when the garbage collector is idle and shared blocks are overwritten or snapshots are created and deleted.

2.2.2 Reference Counting. The alternative to computing reachability asynchronously is to maintain this information synchronously on the data path. The simplest way to do this is to maintain reference counts for all addressable data. When new clones are created, the reference counts of shared data are incremented. When shared data is modified, the reference counts of the original data are decremented and new disk locations are allocated.

With this additional metadata, it is fairly easy to compute how much space a given snapshot or image is consuming: simply scan all the forward references of a given snapshot; any data with a reference count of one is exclusive to the snapshot. While easy to implement, this may still be a costly procedure for large images with many allocated blocks.

Maintaining persistent reference counts can incur significant performance penalties due to metadata write amplification if implemented naively and can lead to serious corruption if implemented incorrectly, so systems that employ this strategy must carefully optimize both their metadata layout and update procedures to mitigate these risks [11, 25, 38]. These optimizations essentially defer some of the cost of maintaining explicit references until such time as they actually need to be read. This somewhat amortizes the performance penalties of reference counting, but it can also lead to stale metadata and/or costly queries. Thus while explicit references make it easier to answer questions about snapshot space consumption than do asynchronous reachability algorithms, they incur heavier costs on the data path and may incur non-trivial query costs once they have been optimized.

3 OWNERSHIP ACCOUNTING

As described in § 2.2, conventional approaches to ownership accounting are based on analyses of *physical* block references. Here we present an alternative approach based on an analysis of *logical* block addresses. This approach has two appealing properties: first, it decouples ownership accounting from underlying snapshot implementations, and second, it admits a tidy formulation of the problem in set-algebraic terms.

As a motivating example, consider a garbage collector attempting to reclaim blocks from the snapshot family depicted in Figure 3. Assume the logical block 0 in node A is mapped to the physical block b. Before the garbage collector can recycle b, it must ensure that no snapshots have the block mapped into their address spaces. One way to do this would be to resolve every logical address in every snapshot to confirm that there are no live references to b.

Another approach, however, would be to observe that the implicit sharing of CoW snapshots guarantees that b will be mapped to the same logical address 0 in all images that reference it. Consequently, any images that have written to logical address 0 no longer reference the original block b, but instead reference new private copies. Further, any image descending from a snapshot that does not reference b will not reference b itself.

In other words, if the path from every live image to the root of the snapshot tree includes a write to logical address 0, then the physical block b is no longer needed and can be recycled. As we demonstrate below, this predicate can be expressed in terms of union and intersection operations over the set of logical addresses written to each node in the snapshot tree, providing a convenient framework for calculating block ownership.

Note that this approach only accounts for implicit sharing due to copy on write techniques. In particular, it does not account for explicit sharing due to deduplication, and it may thus overestimate snapshot sizes in systems that share blocks across independent images. Extending the algorithm to accommodate deduplication is a promising avenue for future research.

3.1 Definitions

We now present a more formal treatment of ownership accounting using logical address analysis.

Given a snapshot tree T with k nodes, let $L(T)$ give the tree's leaf nodes:

$$T = \{n_0, n_1, ..., n_{k-1}\}$$
$$L(T) = \{n \mid n \in T, n \text{ has no children}\}$$

Let $\mathcal{W}_{n_i}$ give the set of all logical addresses written to node n_i, and let $R(n_i)$ give the set of all nodes along the path from n_i to the root of the tree. We say that an address a is *reachable* from n_i if it has been written to any node in $R(n_i)$, and we designate the set of all such addresses $\mathcal{R}_{n_i}$:

$$\mathcal{R}_{n_i} = \bigcup_{n \in R(n_i)} \mathcal{W}_n$$

Let $P(n_i, n_j)$ give the set of all nodes along the path from leaf node n_i to leaf node n_j, *excluding common ancestors*:

$$P(n_i, n_j) = R(n_i) \oplus R(n_j)$$

We call the set of addresses written to nodes in $P(n_i, n_j)$ *divergent* writes, because they break any potential sharing between the two nodes; this set is given by $\mathcal{D}_{<n_i,n_j>}$:

$$\mathcal{D}_{<n_i,n_j>} = \bigcup_{n \in P(n_i,n_j)} \mathcal{W}_n$$

For a leaf node n_i, we designate as $\mathcal{D}^*_{n_i}$ the set of addresses that have diverged in every other leaf node:

$$\mathcal{D}^*_{n_i} = \bigcap_{n_j \in L(T), n_j \neq n_i} \mathcal{D}_{<n_i,n_j>}$$

Finally, we say that address a is *exclusive* to a leaf node n_i if it is reachable from n_i and it has diverged in every other leaf node; the set of all such addresses is given by $\mathcal{E}_{n_i}$:

$$\mathcal{E}_{n_i} = \mathcal{R}_{n_i} \cap \mathcal{D}^*_{n_i}$$

$\mathcal{E}_{n_i}$ represents the storage cost of n_i in the sense that if n_i were to be deleted, only blocks corresponding to these addresses could be reclaimed.

3.2 Computing Uniqueness

Algorithm 1 demonstrates how $\mathcal{E}_{n_i}$ is calculated. First, the set of all addresses reachable from n_i is computed by taking the union of writes to nodes from n_i to the root. Then the set of divergent writes is computed for every other leaf node in the tree. Finally, the intersection over all divergent write sets and all reachable addresses is computed, giving $\mathcal{E}_{n_i}$. Because paths are always traversed in order from n_i to the root or other leaf nodes, redundant union operations can be avoided via simple memoization.

To compute block ownership for a leaf in a tree with N nodes, Algorithm 1 must calculate at most $O(N)$ divergent sets. Because the maximum path length for any given divergent set is also $O(N)$, the computation requires at most $O(N^2)$ union and $O(N)$ intersection operations, respectively.

Algorithm 1 Ownership Accounting

```
1:  procedure EXCLUSIVE(T, n_i ∈ L(T))
2:      p ← n_i
3:      for each node n in PATH(n_i, root) do
4:          table[n] ← W_n ∪ W_p
5:          p ← n
6:      end for
7:      for each leaf node n_j ∈ L(T), n_i ≠ n_j do
8:          for each node n in PATH(n_i, n_j) do
9:              if n ∉ table then
10:                 table[n] ← table[p] ∪ W_n
11:             end if
12:             if n ∈ P(n_i, n_j) then
13:                 p ← n
14:             end if
15:         end for
16:     end for
17:     E_{n_i} ← table[root]
18:     for each leaf node n_j ∈ L(T), n_i ≠ n_j do
19:         E_{n_i} ← E_{n_i} ∩ table[n_j]
20:     end for
21:     return E_{n_i}
22: end procedure
```

3.3 Incremental Updates

The $O(N^2)$ cost of Algorithm 1 can be improved in a couple of common scenarios. First, when computing the uniqueness of *all* nodes in a tree at the same time, redundant work can be avoided by caching all intermediate $\mathcal{D}_{<n_i,n_j>}$ results for n_i, because we know $\mathcal{D}_{<n_j,n_i>}$ will subsequently be needed to compute $\mathcal{E}_{n_j}$, and $\mathcal{D}_{<n_i,n_j>} = \mathcal{D}_{<n_j,n_i>}$.

Second, $\mathcal{E}_n$ values for unmodified nodes can be updated *incrementally* as new writes are applied to writable images. For example, consider the common scenario where periodic backups are retained for a writable image. Let n_w give the writable node and $B = \left\{n_{b_0}, n_{b_1}, ..., n_{b_{k-1}}\right\}$ give the backups. Due to implicit sharing, writes to n_w may affect ownership accounting for possibly many backup nodes, meaning that for a given node $n_b \in B$, $\mathcal{E}_{n_b}$ may need to be recomputed to reflect the changes made to $\mathcal{W}_{n_w}$. Memoized values can be used for most of the component subexpressions of $\mathcal{D}^*_{n_b}$. Specifically, the previously-computed values of $\mathcal{D}_{<n_b,n_{b_j}>}$ for all $n_{b_j} \in B$ can be used because we know that the divergent sets between any two unmodified nodes cannot have changed: $\mathcal{W}_n$ never changes for internal nodes, and by assumption, $\mathcal{W}_{n_b}$ and $\mathcal{W}_{n_{b_j}}$ have not changed either.

With these techniques, the initial $O(N^2)$ cost of Algorithm 1 need only be incurred once when 'priming' the data structures for a given snapshot tree T, and this cost can be further amortized over time as subsequent queries are serviced.

4 OWNERSHIP SKETCHES

Ownership sketches encapsulate the data structures and logic needed to efficiently implement ownership accounting. The key idea of these sketches is to associate *counters* with each image or snapshot in a tree. These counters are used to record the set of logical addresses that have been written for each image in the sketch.

Table 2 presents the interface provided by ownership sketches. An image is initially represented as a single flat address space with an empty counter. As writes are applied to the image, its counter is updated with the logical addresses of these writes. When an image is snapshotted, the original counter is 'frozen' (i.e., made immutable), and a new, empty counter is created for the new image and its snapshot. When an image or snapshot is deleted, its counter is discarded, and its parent node is recursively removed from the tree if it has no remaining children.

At any point in time, the amount of data exclusive to a particular image can be estimated by invoking the `exclusive` routine, which implements Algorithm 1. In this way the overhead of recording writes is kept low (requiring just an update to an in-memory counter), while the cost of estimating $\mathcal{E}_n$ is amortized by deferring more expensive computations until such time as estimates are needed.

Ownership sketches cache intermediate results when appropriate, making it trivial to apply incremental updates. When computing $\mathcal{E}_{n_i}$, the sketch library caches the intermediate sketches corresponding to $\mathcal{D}_{<n_i,n_j>}$ for all $n_j \in L(T)$. When recomputing $\mathcal{E}_{n_i}$ after new writes have been recorded, cached results can be used for any paths between leaf nodes that have not changed. When new writes or snapshot operations are applied to images in the sketch, any cached results affected by the changes are automatically invalidated.

Sketches support pickling, allowing them to be lazily flushed to disk in simple flat files. Lazy flushing means that some updates may be lost in the event of a crash, but this is not a grave concern, because ownership sketches are decoupled from critical system metadata like bitmaps and reference counts, which have much more demanding consistency requirements.

4.1 System Integration

The sketch interface is designed to be easily retrofitted into existing storage systems. To this end it models relevant storage operations – `create`, `delete`, `clone`, and `write` – in a generic manner that is equally compatible with read-only and read-write snapshots for both volume- and file-based implementations. Integration is as simple as instrumenting these four operations; no other modifications are required. In particular, existing on-disk metadata need not be changed.

Sketches are generally updated only in response to user actions, but they may be queried by any number of subsystems. For instance, UI modules may wish to present space usage statistics to users, while garbage collection and quota enforcement modules may use sketches to prioritize scheduling.

For example, typical mark and sweep garbage collectors perform periodic scans of all metadata to identify blocks that are no longer referenced. Without intelligent scheduling, this can lead to a substantial amount of wasted work scanning large numbers of live blocks. An obvious optimization is to prioritize scanning recently deleted images and snapshots under the assumption that their blocks may be reclaimed. However, this assumption may be invalid if these blocks are still shared by live snapshots. A simple refinement is to use ownership sketches to prioritize scanning deleted images that have a large number of exclusive blocks, because these are known to be reclaimable. Scanning images and snapshots with few exclusive blocks can be deferred until related snapshots are deleted as well.

Note that for this use case, ownership sketches need not indicate *which* blocks may be reclaimed – merely indicating *how many* blocks are reclaimable provides enough information to prioritize scheduling. Garbage collectors can rely on existing algorithms and data structures to locate reclaimable blocks while using ownership sketches to avoid scanning low-value candidates, effectively pruning the search space. Because existing system metadata provides the authoritative record of liveness in this approach, the results computed by ownership sketches need not be perfectly accurate to be useful.

5 COUNTING

Algorithm 1 produces the complete set of logical addresses that are exclusive to a single snapshot. By resolving these addresses to physical blocks, the actual storage locations can be identified. But, as mentioned above, in many scenarios it is useful to have a rough idea of *how many* blocks are exclusive to a given image before going to the trouble of identifying these blocks. In such cases, it suffices to know the size or *cardinality* of $\mathcal{W}_n$ rather than knowing its constituent elements.

A practical ownership sketch implementation thus requires counters capable of calculating both individual cardinalities and cardinalities of set expressions over multiple counters. We now describe three different counting techniques that satisfy these requirements and discuss some of the trade-offs involved in choosing one over the other.

5.1 Perfect Counting

A naive counter implementation could simply use flat bitmaps to record which logical blocks have been written to a particular image. This approach is nice because it is simple and in many cases, it could leverage existing metadata maintained by underlying storage implementations.

However, flat bitmaps, while common, are not the most efficient data structure available for perfect counters, and they can incur significant overhead [23]. For example, given a 100 GiB image with 32 snapshots, ownership metadata must be maintained for 65 nodes. Assuming images are stored as VHDs [46], we could use the sector-granularity bitmaps (i.e., one bit per 512 bytes of data) already maintained internally by this file format for ownership accounting.

Method	Description
`create(name) → sketch`	create a new sketch for a base image with the given name
`get(sketch, name) → image`	return a reference to the given image
`clone(sketch, image, name) → image`	create and return a clone of the given image
`delete(sketch, image)`	remove an image (or clone) from the sketch
`update(sketch, image, block)`	record a write to the image at the given block
`exclusive(sketch, image) → cardinality`	return an estimate of the image's unique blocks
`pack(sketch) → byte array`	serialize a sketch
`unpack(byte array) → sketch`	deserialize a sketch

Table 2: Ownership Sketch API

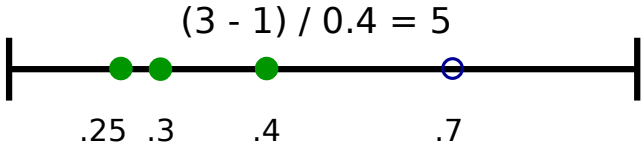


Figure 4: KMV sketches record the k the smallest hashed values of a data set (shown in green) to estimate the cardinality of the entire multiset (including unfilled blue dots).

Doing so, however, could require reading and comparing as much as 1.5 GiB of bitmaps in order to determine how much data is exclusive to a single snapshot. While this approach may suffice for smaller images (or systems with larger block sizes), it scales poorly in more demanding environments.

Instead, we leverage *roaring bitmaps* [4], an efficient, sparse, compressed bitmap format commonly used to implement bitmap indexes for databases. This data structure is particularly well-suited for ownership accounting because it provides efficient union and intersection implementations thanks to its ability to support random access, and it commonly achieves significant space savings over naive bitmap representations.

5.2 Probabilistic Counting

For write-heavy workloads or large snapshot trees, the overhead required to maintain perfect counters may become excessively onerous. In these situations, we can provide approximate ownership accounting by leveraging special probabilistic data structures. This approach draws from a large body of existing research in streaming algorithms that investigates how to estimate the cardinality of large data sets with very low memory overheads. We now present a brief survey of existing cardinality sketches and detail some of the challenges we face in applying them to the problem of ownership accounting.

5.2.1 Cardinality Sketches. Streaming algorithms are so named because they are designed to process very large data sets in a single pass, typically with the constraint that time and space overheads must be very small (e.g., sublinear) with respect to the size of the input data set. The problem of counting distinct elements is well-studied within this paradigm [7, 9, 15, 16, 19, 24, 49]. The severe memory constraints in this domain typically lead to *probabilistic* estimates; indeed, as Flajolet et al. remark in a seminal paper on the subject, "all known efficient cardinality estimators rely on randomization, which is ensured by the use of *hash* functions" [16].

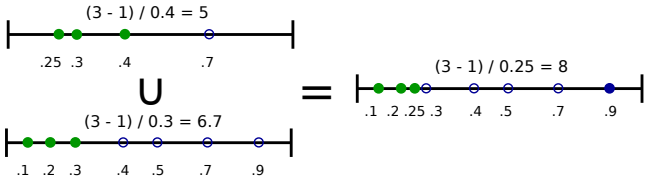
Many estimation techniques involve constructing *sketches* or *synopses*, compact representations of data streams from which accurate cardinality estimates can be computed. Sketches are updated as elements of the data stream are processed, and at any point in time, specially-designed *estimators* can be applied to the sketches to calculate approximate cardinalities.

A wide variety of sketches have been studied, with HyperLogLog (HLL) [16] representing the current state of the art. For expository purposes, however, we will consider the simpler K Minimum Values (KMV) sketch [2]. The basic idea behind this sketch is to use a strong hash function to map elements of the input stream to a uniform random distribution with a fixed domain. Intuitively, the probability of observing a very small value in the sequence of hashed values increases with the number of unique elements encountered. In fact, a cardinality estimate can be derived from *only* the smallest observed value, although the variance of such an estimator would be quite large due to the likelihood of outliers.

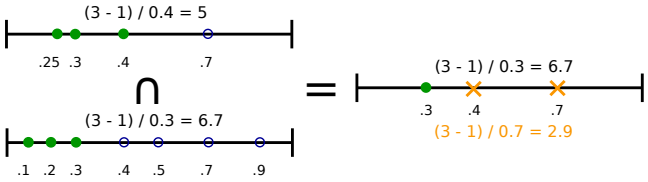
In practice, KMV provides more accurate estimates by maintaining, as its name suggests, the k smallest hashed values. Assuming these values are drawn from a uniform random distribution, an estimate of the cardinality of the input stream can be derived from the average distance between them, as shown in Figure 4. An unbiased cardinality estimator for this sketch is given by $(k-1)/\max(KMV)$, where $\max(KMV)$ gives the largest hash value recorded in the sketch [3].

The KMV sketch is known as an (ϵ, δ)-approximation, meaning that $Pr[|D' - D| <= (1-\epsilon) * D] >= 1-\delta$, where D and D' give the actual and estimated cardinalities, respectively. Given a sufficiently large input stream, increasing k provides increasingly accurate estimates, at the cost of additional storage and processing overhead. For example, for $\delta = 0.05$ and $k = 1024$, the theoretical upper bound of ϵ is roughly 0.06127 as the size of the input stream approaches infinity, and was observed to be around 0.06124 for a real data set of one million elements [3].

5.2.2 Set Operations on Cardinality Sketches. An appealing property of KMV sketches is that multiple sketches over separate data streams can be combined to produce an estimate of the cardinality



(a) The union operation is applied to KMV sketches by taking the k smallest values from the merged sketches. The resulting sketch is identical to one which would have been created by counting the union of the two input data sets.



(b) The intersection operation is applied to KMV sketches by taking all values present in *both* sketches. The error bounds of this technique are proportional to the cardinality of the larger set. In this example, the second counter has observed more values, leading it to "forget" values (shown as orange crosses) that would have been recorded by a sketch counting the intersection of the two input data sets.

Figure 5: Set operations on KMV sketches

of the *union* of the original streams – in other words, an analogue of the set union operation can be applied directly to sketches themselves. This is valuable in scenarios where it is infeasible or impossible to maintain sketches for all combinations of data streams that might subsequently be of interest.

The set union operation is trivially applied to two KMV sketches by constructing a new sketch with the k smallest values present in either of the original sketches; as Figure 5a shows, this produces exactly the same sketch as one that would be produced by processing the union of the input data streams. Indeed, many common sketches, including HLLs, naturally lend themselves to union estimates in a similar manner.

Unfortunately, many sketches do *not* support set intersection operations in quite the same way. Conceptually, the set intersection operation can be applied to two KMV sketches by constructing a new sketch with the k smallest values present in *both* original sketches. However, this is not necessarily the same sketch that would have been produced by processing the intersection of the original streams, as can be seen in Figure 5b. Indeed, it is likely that two sketches combined in this way will produce a new sketch with fewer observations than the originals, leading to less accurate cardinality estimates.

Some of the more sophisticated sketches like HLLs are structured such that no obvious set intersection analogue has been identified for them at all. For these sketches, estimates based on the *inclusion/exclusion principle* have been proposed as an alternative to directly estimating intersection cardinalities. This approach is based on the fact that for two sets A and B, $|A \cap B| = |A \cup B| - |A| - |B|$. However, this proposal, as well as related techniques based on estimating Jaccard coefficients, produce estimates with errors proportional to the size of the union of the two sets. This is problematic when the cardinality of the intersection is much smaller than that of the union, as the error bounds may be larger than the estimates themselves.

This presents a challenge when using cardinality sketches for ownership accounting, because, as shown in § 3.1, computing $\mathcal{E}_{n_i}$ involves computing the intersection of the divergent sets between n_i and all other leaf nodes. In situations where $\mathcal{D}^*_{n_i} \ll \mathcal{D}_{<n_i,n_j>}$, the estimates produced by this technique may have large relative errors. This is somewhat mitigated by the fact that the storage costs for snapshots in such situations are by definition relatively low, since $|\mathcal{E}_{n_i}|$ will be small.

Fortunately, a handful of promising new techniques have been proposed for estimating the cardinality of arbitrary set expressions [6, 14, 18, 44]. These approaches leverage more statistically efficient estimators and generally provide more accurate results. We describe three probabilistic counters based on these techniques in § 6.

5.3 Hybrid Counting

The low, fixed cost of probabilistic counters suggests a third, hybrid counting strategy wherein perfect counters are augmented with probabilistic counters. In this scheme, both types of counters are maintained for every image in an ownership sketch. For images which see only small numbers of writes, the perfect counters can be used to provide exact ownership accounting. However, if the storage costs of maintaining perfect counters becomes too burdensome, the sketch can convert to using probabilistic counters (with fixed storage overheads) to provide approximate ownership accounting.

6 IMPLEMENTATION

We have implemented the interface presented in Table 2 as a generic Python library that manipulates an abstract counter interface. As we demonstrate in § 7, supporting different counter implementations allows users to trade accuracy for overhead according to their needs. The library implements Algorithm 1 by constructing set-algebraic *expressions* in conjunctive normal form. Expressions are simplified as much as possible using symbolic analysis before being passed to counter implementations.

Counter implementations expose a simple interface for counting values and computing cardinality estimates of both individual counters and compound expressions over multiple counters. They may also optionally provide an interface for constructing new counters corresponding to compound expressions. The sketch library uses these constructed counters to reduce the cost of incremental updates when possible. For example, counters constructed from the divergent sets of all read-only snapshot pairs are cached by the sketch library and substituted for equivalent compound expressions in subsequent computations, thereby avoiding redundant union operations.

We now present a few salient details about the four counters we have implemented in order to provide more intuition about how they can be expected to perform as part of ownership sketches.

roaring. The internal structure of roaring bitmaps is such that they can only represent 32-bit values. We support 64-bit disk addresses (or 52-bit block addresses, assuming 4 KiB blocks) by combining multiple bitmaps in a hash table keyed by the most significant 32 bits of the address space. Updating a `roaring` counter involves first locating the correct bitmap in the hash table, and then updating the bitmap itself. Union and intersection operations are performed by iterating hash tables in sorted order and comparing corresponding bitmaps in a piece-wise manner.

kmv. Our baseline probabilistic sketch uses *k-minimum values* cardinality counters [3] augmented with improved union and intersection operators [44]. Our implementation maintains the k values in a balanced tree. Updating the sketch involves hashing the original value to generate a uniform random 32-bit word that is inserted into the tree only if it is smaller than the largest stored value. The union operation involves constructing a new balanced tree using the k smallest values found in both original trees, and the intersection operation requires constructing a new tree composed of the intersection of the values from both original trees.

skmv. Our second probabilistic sketch uses *streaming kmv* sketches [44]. These sketches extend the original `kmv` sketch with a running cardinality estimate; their update procedure is similar to that of the `kmv` sketch, except that the running estimate is updated when new k-minimum values are inserted into the sketch. Union and intersection operations involve first applying these operations to the underlying `kmv` sketches, and then estimating a final cardinality by combining weighted estimates from each component sketch with the estimate produced by the final combined sketch. Unlike `kmv` sketches, `skmv` sketches notably do *not* support closed union and intersection operations (i.e., they can estimate union and intersection cardinalities, but they cannot produce new union and intersection sketches), because the running estimates of the individual sketches cannot meaningfully by combined to produce a new tally for the resulting sketch.

union. Our final probabilistic sketch uses a continuous variant of Flajolet Martin (FM) sketches [17] combined with the *proportional union* technique [6] for estimating the cardinality of set operations. The `union` sketch uses stochastic averaging to update a fixed-size array of 32-bit words. Updating the sketch involves hashing the original value twice with two independent hash functions; the first hashed value is used as an index into the array of words, and the second value is inserted at the designated index if it is smaller than the existing value stored there. The union operation, which is closed for this sketch, involves constructing a new array where each element is the minimum of the corresponding elements in the original arrays. Estimating the cardinality of union and intersection expressions additionally requires parameter estimation based on a simple calculation inspired by the maximum likelihood estimation technique.

7 EVALUATION

In this section we investigate two high-level properties of ownership sketches. First, we measure the cost of maintaining and querying four different sketch variants and show that our technique meets the performance requirements of modern storage systems. Second, we study the accuracy of probabilistic sketches as applied to real-world storage traces, and we present a parametric analysis of the primary contributor to estimation error using a synthetic workload generator.

Unless otherwise noted, all probabilistic counters evaluated in the following experiments are configured to use 256 KiB of memory (`roaring` counters are unbounded in size and grow proportionally to the number of unique values encountered). All experiments were performed on a 3.3 GHz Intel Core i5 CPU with 8 GiB of RAM. Confidence intervals are computed using bootstrapping.

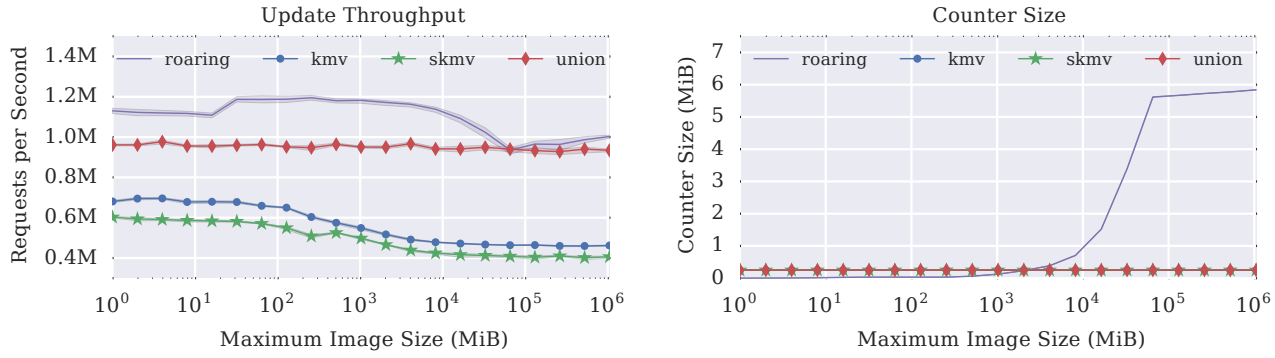
7.1 Performance

Figure 6 shows the throughput and memory use of our four sketch variants. In both experiments, we measure the time required to update a sketch with one million values drawn from a uniform random distribution. We vary the sample space across trials in order to generate value sequences with different ranges and cardinalities.

As Figure 6a shows, every sketch variant is capable of processing hundreds of thousands of updates per second. The `union` sketch exhibits a constant throughput across the spectrum of value ranges because the cost of updating this sketch is $O(1)$. The `kmv` and `skmv` sketches show a roughly thirty percent decrease in throughput as the input range grows from 25 K to 2.5 M because an increasing number of new k-minimum values are encountered within these ranges, leading to more frequent and costly tree updates until a steady state is reached. The `skmv` sketch shows an additional overhead due to the calculations involved in maintaining its running estimate. Finally, the `roaring` sketch shows very good throughput across the input ranges, with a dip where the range spills over what can be accounted for with a single bitmap.

Figure 6b shows how the size of `roaring` sketches scales both with cardinality *and* input range. A pronounced increase in memory overhead can be seen once the range of input values exceeds 4 M, as multiple sparse bitmaps must be allocated to track values larger than 2^{32}. The overhead levels off because the number of updates is restricted to one million; recording more updates at larger input ranges would require additional memory. This figure also illustrates the benefit of the probabilistic sketches, which, as expected, require a constant amount of memory regardless of input range and cardinality.

Figure 7 shows the time required to compute the number of blocks unique to a given image in a snapshot tree as a function of the size of the tree. For each observation, we construct a snapshot tree with the given number of nodes and record 256 K writes to each node, with write values drawn from a uniform random distribution. Then we measure the time required to perform a *full* uniqueness query that computes the number of blocks unique to a given leaf node. As described in § 3.2, this requires on the order of $O(N^2)$



(a) Update Throughput **(b) Memory Use**

Figure 6: Performance and memory consumption of four different sketch implementations. For each observation, we record the time and space required to update a sketch with a sequence of one million values, drawn from a uniform random distribution with the sample space restricted to the maximum image size. We present the average of ten observations per image size, with bands giving 98% confidence intervals.

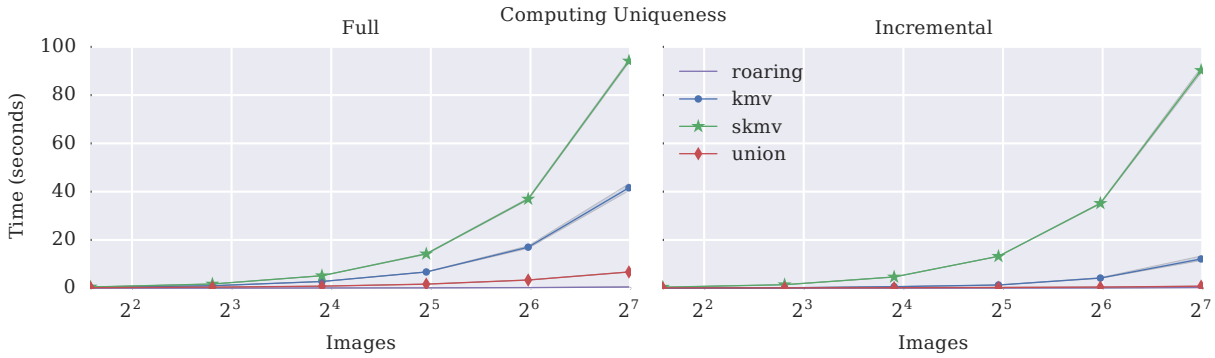


Figure 7: Uniqueness queries against trees of varying sizes. *Full* queries involve computing the cardinality of all paths in the tree; *incremental* queries need only compute the cardinality of paths affected by sketch updates. We plot the average of five queries against each tree size, with bands giving 98% confidence intervals.

union operations for a tree with N leaves, but as the figure shows, the efficiency of the specific sketch algorithm has a big impact on overall computation time.

In particular, the `roaring` sketch benefits from an optimized C implementation that provides a very fast union implementation. The `kmv` sketch, whose union operator (implemented in pure Python) involves manipulating balanced trees, shows significantly more overhead than the `union` sketch, whose union operator only needs to iterate over flat arrays. Finally, the `skmv` sketch shows the highest overhead because it does not support a closed union operation (i.e., it can compute cardinality estimates for union operations, but not the corresponding sketches). Consequently, redundant union operations for common subpaths cannot be memoized (as described in § 3.2), but must instead be recomputed for each path.

We also measure the time required to perform an *incremental* uniqueness query. We do this by first issuing a *full* query for a leaf node n_i to prime the sketch, then applying writes to a leaf node n_j, and finally measuring the time to service a second query for n_i. As explained in § 3.3, incremental queries against sketches that support a closed union operator can be partially satisfied by cached sketches for paths not affected by new updates. This is particularly useful in common scenarios where only one or a few "live" images in a snapshot tree are actively updated at any given time. This optimization significantly reduces query time for both the `kmv` and `union` sketches, but not the `skmv` sketch (again because it does not support a closed union operator).

7.2 Accuracy

We now turn our attention to the accuracy of probabilistic sketches. As mentioned in § 5.2.2, many probabilistic sketches provide accurate estimates of the cardinality of union operations but may not do as well when estimating the cardinality of intersection operations, especially when the cardinalities of the individual sketches differ significantly.

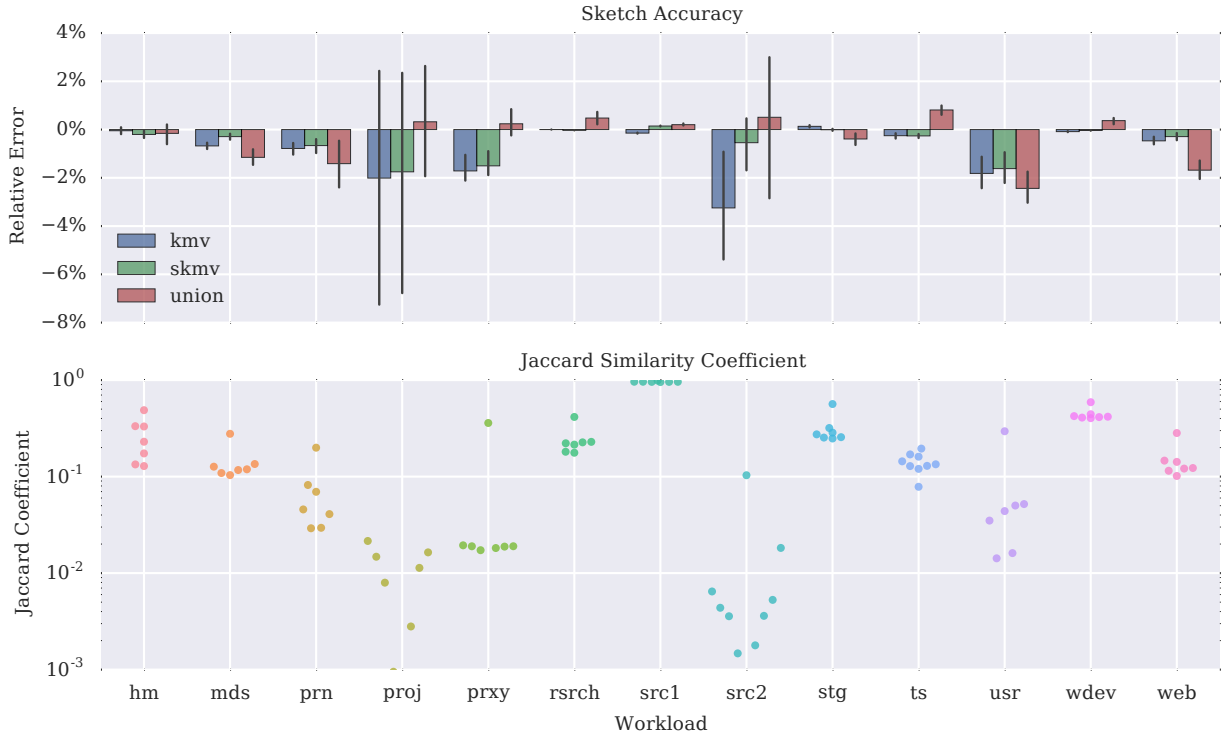


Figure 8: Sketch accuracy and Jaccard Coefficients for the MSR Cambridge workloads.

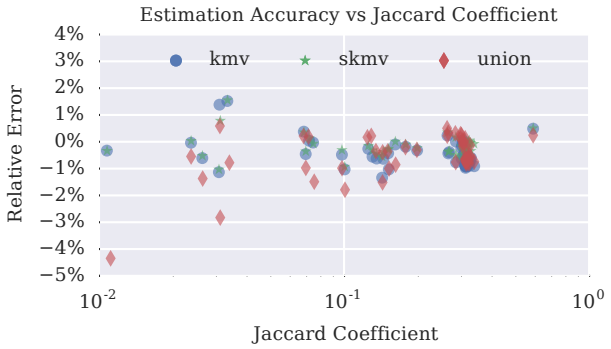


Figure 9: Sketch accuracy and Jaccard Coefficients for the FIU workloads.

Intuitively, we would expect approximate ownership sketches to provide accurate estimates when all images in a tree have absorbed roughly the same number of unique writes. This is likely to be the case for workloads with regular write patterns that are snapshotted at regular intervals. Whether and to what degree real workloads exhibit this property will largely determine the accuracy of probabilistic sketches.

Figure 8 presents the accuracy of estimates for a collection of server-class storage traces collected at MSR Cambridge [34]. The traces record a week's worth of block-level storage requests for a variety of servers, including revision control servers, a filer, and a web server. We replay each trace with a simulated daily snapshot schedule against each of our probabilistic sketches. We then compute the number of blocks that are unique to each image at the end of the trace, and compute the average (and 98% confidence intervals) of the error of these estimations relative to the corresponding ground truth values.

As the top half of Figure 8 shows, the probabilistic estimates are quite accurate for these workloads, with relative errors generally falling within ±4% of the actual values. These error rates are likely to be tolerable in a number of practical scenarios. For example, they are unlikely to vitiate the priority rankings that a garbage collector might assign in an effort to avoid scanning low-value snapshots. In general, the `skmv` and `union` sketches achieve better accuracy than the basic `kmv` sketch, although no single sketch outperforms all others for all traces.

The bottom half of Figure 8 presents a rough measure of the expected error for these workloads. As mentioned in § 5.2.2, the error bounds for set intersection cardinality estimates are proportional to the cardinality of the union of the constituent sets. This proportion is given by the *Jaccard similarity coefficient*, which, for a leaf image n_i of a sketch of tree T, we define as $|\mathcal{E}_{n_i}|/|\mathcal{R}_{n_i} \cup \mathcal{U}^*_{n_i}|$, where $\mathcal{U}^*_{n_i}$

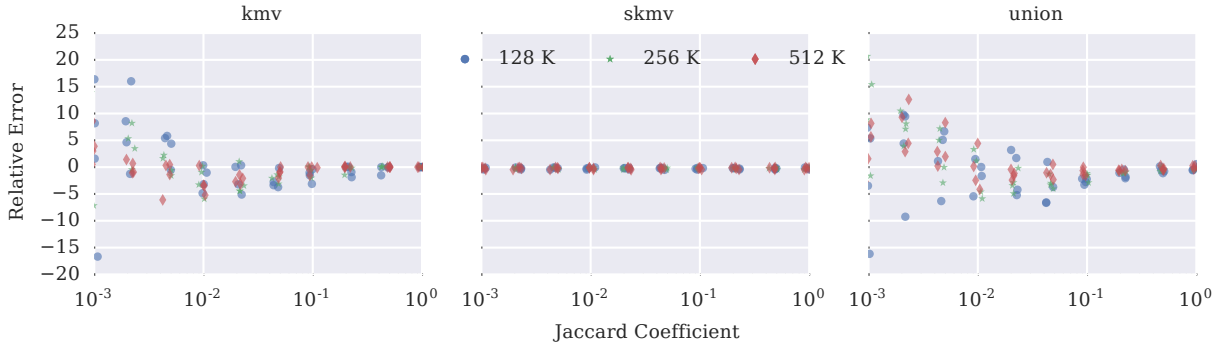


Figure 10: Sketch accuracy and Jaccard Coefficients for sketches of varying sizes applied to synthetic data sets.

is the union of divergent writes between n_i and all other leaves:

$$\mathcal{U}^*_{n_i} = \bigcup_{n_j \in L(T), n_j \neq n_i} \mathcal{D}_{<n_i, n_j>}$$

The bottom half of Figure 8 plots these values for each snapshot of each workload. The figure shows a clear correlation between Jaccard coefficients and sketch accuracy. In particular, the workloads `proj` and `src2`, which exhibit the largest relative errors, also exhibit the smallest Jaccard coefficients; both workloads have multiple snapshots with very few unique blocks relative to the other snapshots in their sketches, leading to higher relative errors.

This suggests that approximate ownerhip accounting is best suited for workloads exhibiting only moderate write variance accross snapshots; differences of more than two or three orders of magnitude between snapshots may lead to unreasonable estimation errors. It is thus natural to wonder how frequently "common" workloads satisfy this criterium.

As an additional data point, Figure 9 plots the Jaccard Coefficients and relative errors of sketches derived from a second set of publicly available storage traces [26]. These traces record three weeks of IO requests to three different production servers from a large university. We again simulate daily snapshot schedules when replaying these traces, plotting the estimated uniqueness cardinalities at the end of the trace. Each data point gives the estimation accuracy for a particular snapshot plotted against its Jaccard coefficient. Similar to the MSR traces, the relative error rates for these traces are within reasonable bounds, with the largest errors occurring when Jaccard similarity is lowest.

Finally, we systematically measure the effect of sketch size on estimation accuracy for varying Jaccard similarities. Figure 10 plots the relative errors for estimated intersection cardinalities between two counters. For each experiment, we construct two synthetic data sets such that the cardinality of their intersection falls between 4 K and 32 K and their Jaccard Coefficients vary between 10^{-3} and 1. We report the estimated intersection cardinalities relative to the actual values for sketches with sizes ranging from 128 KiB to 512 KiB. In general, the accuracy of all sketch variants converges as the Jaccard coefficient increases, and larger sketches are slightly more accurate across the spectrum. The `skmv` sketch shows particular robustness for all coefficients in this experiment.

8 CONCLUSION

In storage systems, maintaining metadata that accounts for the ownership of stored data is critical for reporting and charging for space consumption, enforcing quotas, and reclaiming data that is no longer referenced. The addition of space-efficient snapshots greatly complicates accounting as an aspect of system design because accounting relationships between data and objects become multi-dimensional. As a result, snapshot implementations must choose to either tolerate the added data-path complexity of maintaining these relationships as first-class storage metadata or cope with the staleness of information that comes from deferring ownership accounting to background tasks.

In this work, we advocate a third approach. We observe that accounting queries can be expressed as set cardinality queries between object snapshots. In light of this, we introduce a space- and compute-efficient hybrid data structure that can be used to track ownership within each snapshot. The data structure uses probabilistic sketches to estimate the amount of uniqueness between snapshots, which in turn makes it possible to report per-snapshot capacity consumption, enforce usage quotas, and identify appropriate opportunities for garbage collection.

By combining sketches with roaring bitmaps, we demonstrate an opportunity to precisely account ownership at small to moderate snapshot sizes, and to remain compact while still accurately accounting for consumption as snapshots become large. Our implementation decouples accounting from the rest of the storage implementation and so is easily portable to new systems.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees and our shepherd, Vijayan Prabhakaran, for the very helpful feedback they provided. We would also like to thank Steven Smith and Stephen Ingram for the invaluable roles they played in the conception of this work.

REFERENCES

[1] Alain Azagury, MF Factor, Julian Satran, and William Micka. 2002. Point-in-time Copy: Yesterday, Today and Tomorrow. In *Tenth Goddard Conference on Mass Storage Systems and Technologies*. NASA; 1998, 259–270.

[2] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. 2002. Counting Distinct Elements in a Data Stream. In *Proceedings of the 6th International Workshop on Randomization and Approximation Techniques (RANDOM)*. Springer-Verlag, Cambridge, Ma, USA, 1–10.

[3] Kevin Beyer, Peter J Haas, Berthold Reinwald, Yannis Sismanis, and Rainer Gemulla. 2007. On Synopses for Distinct-value Estimation Under Multiset Operations. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. ACM, 199–210.

[4] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. 2016. Better Bitmap Performance with Roaring Bitmaps. *Software, Practice and Experience* 46, 5 (2016), 709–719.

[5] Ramesh Chandra, Nickolai Zeldovich, Constantine Sapuntzakis, and Monica S. Lam. 2005. The Collective: A Cache-based System Management Architecture. In *Proceedings of the Second Conference on Symposium on Networked Systems Design and Implementation*. 259–272.

[6] Aiyou Chen, Jin Cao, and Tian Bu. 2007. A Simple and Efficient Estimation Method for Stream Expression Cardinalities. In *VLDB*. ACM, 171–182.

[7] Aiyou Chen, Jin Cao, Larry Shepp, and Tuan Nguyen. 2011. Distinct Counting with a Self-learning Bitmap. *CoRR* abs/1107.1697 (2011).

[8] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E Platform: Design, Implementation, and Applications. *ACM Transactions on Computer Systems* 30, 1 (Feb. 2012), 2:1–2:49.

[9] Graham Cormode, Mayur Datar, Piotr Indyk, and S. Muthukrishnan. 2003. Comparing Data Streams Using Hamming Norms (How to Zero In). *IEEE Transactions on Knowledge and Data Engineering* 15, 3 (2003), 529–540.

[10] Disk space usage of zfs snapshots and filesystems 2008. (2008). Retrieved August 10, 2017 from http://markmail.org/thread/o5pffx7ltfpyu7rj

[11] Chris Dragga and Douglas J. Santry. 2016. GCTrees: Garbage Collecting Snapshots. *ACM Transactions on Storage* 12, 1, Article 4 (Jan. 2016), 32 pages.

[12] James R Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. 1986. Making Data Structures Persistent. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*. ACM, 109–121.

[13] John K. Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A. Smith, and Edward R. Zayas. 2008. FlexVol: Flexible, Efficient File Volume Virtualization in WAFL. In *USENIX Annual Technical Conference*. USENIX Association, 129–142.

[14] Otmar Ertl. 2017. New Cardinality Estimation Algorithms for HyperLogLog Sketches. *CoRR* (2017).

[15] Cristian Estan, George Varghese, and Mike Fisk. 2003. Bitmap Algorithms for Counting Active Flows on High Speed Links. In *Internet Measurement Comference*. ACM, 153–166.

[16] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. Hyperloglog: the Analysis of a Near-optimal Cardinality Estimation Algorithm. In *Analysis of Algorithms 2007 (AofA07)*. 127–146.

[17] Philippe Flajolet and G. Nigel Martin. 1985. Probabilistic Counting Algorithms for Data Base Applications. *J. Comput. System Sci.* 31, 2 (1985), 182–209.

[18] Sumit Ganguly, Minos N. Garofalakis, and Rajeev Rastogi. 2004. Tracking Set-expression Cardinalities Over Continuous Update Streams. *VLDB* 13, 4 (2004), 354–369.

[19] Frédéric Giroire. 2009. Order Statistics and Estimating Cardinalities of Massive Data Sets. *Discrete Applied Mathematics* 157, 2 (2009), 406–427.

[20] Russell J Green, Alasdair C Baird, and J Christopher Davies. 1996. Designing a Fast, On-line Backup System for a Log-structured File System. *Digital Technical Journal* 8 (1996), 32–45.

[21] High IO read load on RHEL 7.3 from thin_ls 2016. (2016). Retrieved August 10, 2017 from https://bugzilla.redhat.com/show_bug.cgi?id=1405347

[22] How to view snapshots and determining the size of snapshots 2016. (2016). Retrieved August 10, 2017 from https://kb.netapp.com/support/s/article/ka31A0000000mrcQAA/how-to-view-snapshots-and-determine-the-size-of-snapshots

[23] Saurabh Kadekodi and Shweta Jain. 2010. Taking Linux Filesystems to the Space Age: Space Maps in Ext4. In *Linux Symposium*. 121–132.

[24] Daniel M. Kane, Jelani Nelson, and David P. Woodruff. 2010. An Optimal Algorithm for the Distinct Elements Problem. In *PODS*. ACM, 41–52.

[25] Ram Kesavan, Rohit Singh, Travis Grusecki, and Yuvraj Patel. 2017. Algorithms and Data Structures for Efficient Free Space Reclamation in WAFL. In *FAST*. USENIX Association, 1–14.

[26] Ricardo Koller and Raju Rangaswami. 2010. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. *ACM Transactions on Storage* 6, 3 (2010).

[27] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. 2009. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*. 1–12.

[28] Edward K. Lee and Chandramohan A. Thekkath. 1996. Petal: Distributed Virtual Disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*. 84–92.

[29] LVM Thin: Integrate thin_ls features into LVM 2016. (2016). Retrieved August 10, 2017 from https://bugzilla.redhat.com/show_bug.cgi?id=1306717

[30] Peter Macko, Margo I. Seltzer, and Keith A. Smith. 2010. Tracking Back References in a Write-Anywhere File System. In *FAST*. USENIX, 15–28.

[31] John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. In *Communications of the ACM*, Vol. 3. ACM, New York, NY, USA, 184–195.

[32] Mark McLoughlin. 2008. The QCOW2 Image Format. (2008). Retrieved August 10, 2017 from https://people.gnome.org/~markmc/qcow-image-format.html

[33] Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield. 2008. Parallax: Virtual Disks for Virtual Machines. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (Eurosys '08)*. 41–54.

[34] Dushyanth Narayanan, Austin Donnelly, and Antony I. T. Rowstron. 2008. Write Off-Loading: Practical Power Management for Enterprise Storage. In *FAST*. USENIX, 253–267.

[35] Problems with btrfs quota 2016. (2016). Retrieved August 10, 2017 from https://redd.it/4qz1qd

[36] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. 2005. Rx: Treating Bugs as Allergies—A Safe Method to Survive Software Failures. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*. 235–248.

[37] Request for feature: disk space usage by lvm snapshot 2015. (2015). Retrieved August 10, 2017 from https://www.redhat.com/archives/linux-lvm/2015-November/msg00020.html

[38] Ohad Rodeh. 2008. B-trees, Shadowing, and Clones. *ACM Transactions on Storage* 3, 4 (2008).

[39] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-Tree Filesystem. *ACM Transactions on Storage* 9, 3 (2013), 9.

[40] Ohad Rodeh and Avi Teperman. 2003. zFS - A Scalable Distributed File System Using Object Disks. In *IEEE Symposium on Mass Storage Systems*. IEEE Computer Society, 207–218.

[41] Mendel Rosenblum and John K. Ousterhout. 1991. The Design and Implementation of a Log-Structured File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP '91)*. ACM, 1–15. Operating System Review 25(5).

[42] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. 1999. Deciding When to Forget in the Elephant File System. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP '99)*. ACM, 110–123. Operating System Review 33(5).

[43] Snapshot destruction making IO extremely slow 2017. (2017). Retrieved August 10, 2017 from https://www.spinics.net/lists/linux-btrfs/msg65796.html

[44] Daniel Ting. 2016. Towards Optimal Cardinality Estimation of Unions and Intersections with Sketches. In *22nd SIGKDD Conference on Knowledge Discovery and Data Mining*. ACM, 1195–1204.

[45] Very slow balance with btrfs-transaction 2017. (2017). Retrieved August 10, 2017 from https://www.spinics.net/lists/linux-btrfs/msg62610.html

[46] VHD Specifications 2017. (2017). Retrieved August 10, 2017 from https://www.microsoft.com/en-us/download/details.aspx?id=23850

[47] VMDK Specifications 2011. (2011). Retrieved August 10, 2017 from https://www.vmware.com/support/developer/vddk/vmdk_50_technote.pdf

[48] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. 2005. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*. 148–162.

[49] Kyu-Young Whang, Brad T. Vander-Zanden, and Howard M. Taylor. 1990. A linear-time Probabilistic Counting Algorithm for Database Applications. *ACM Transactions on Database Systems* 15, 2 (1990), 208–229.

[50] What's eating my space? Missing Snapshots? 2009. (2009). Retrieved August 10, 2017 from http://markmail.org/thread/2btnhnd3d3ctojx3

[51] ZFS snapshot used space question 2012. (2012). Retrieved August 10, 2017 from http://markmail.org/thread/msfiihbjjtsphypr