BigArchive: Low-cost and Scalable Deduplication Storage for VM Snapshot

ABSTRACT

This paper proposes a VM snapshot storage architecture which adopts multiple-level selective deduplication to bring the benefits of fine-grained data reduction into cloud backup storage systems. In this work, we describe our working snapshot system implementation, and provide early performance measurements for both deduplication impact and snapshot operations.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

ACM proceedings, LaTeX, text tagging

1. INTRODUCTION

In a virtualized cloud environment such as ones provided by Amazon EC2[2] and Alibaba Aliyun[1], each instance of a guest operating system runs on a virtual machine, accessing virtual hard disks represented as virtual disk image files in the host operating system. Because these image files are stored as regular files from the external point of view, backing up VM's data is mainly done by taking snapshots of virtual disk images.

A snapshot preserves the data of a VM's file system at a specific point in time. VM snapshots can be backed up incrementally by comparing blocks from one version to another and only the blocks that have changed from the previous version of snapshot will be saved [3, **?**].

Frequent backup of VM snapshots increases the reliability of VM's hosted in a cloud. For example, Aliyun, the largest cloud service provider by Alibaba in China, provides automatic frequent backup of VM images to strengthen the reliability of its service for all users. The cost of frequent backup of VM snapshots is high because of the huge storage demand. Using a backup service with full deduplication support [5, 7] can identify content duplicates among snapshots to remove redundant storage content, but the weakness is that it either adds the extra cost significantly or competes computing resource with the existing cloud services. In addition, data dependence created by duplicate relationship among snapshots adds the complexity in fault tolerance management, especially when VMs can migrate around in the cloud.

Unlike the previous work dealing with general file-level backup and deduplication, our problem is focused on virtual disk image backup. Although we treat each virtual disk as a file logically, its size is very large. On the other hand, we need to support parallel backup of a large number of virtual disks in a cloud every day. One key requirement we face at Alibaba Aliyun is that VM snapshot backup should only use a minimal amount of system resources so that most of resources is kept for regular cloud system services or applications. Thus our objective is to exploit the characteristics of VM snapshot data and pursue a cost-effective deduplication solution. Another goal is to decentralize VM snapshot backup and localize deduplication as much as possible, which brings the benefits for increased parallelism and fault isolation.

By observations on the VM snapshot data from production cloud, we found snapshot data duplication can be easily classified into two categories: *inner-VM* and *cross-VM*. Inner-VM duplication exists between VM's snapshots, because the majority of data are unchanged during each backup period. On the other hand, Cross-VM duplication is mainly due to widely-used software and libraries such as Linux and MySQL. As the result, different VMs tend to backup large amount of highly similar data.

With these in mind, we have developed a distributed multi-level solution to conduct segment-level and block-level inner-VM deduplication to localize the deduplication effort when possible. It then makes cross-VM deduplication by excluding a small number of popular common data blocks from being backed up. Our study shows that common data blocks occupy significant amount of storage space while they only take a small amount of resources to deduplicate. Separating deduplication into multi levels effectively accomplish the major space saving goal compare the global complete dedu-

	Scalability	Low-cost	Full Dedup
DDFS	N	N	Y
Ex-bin	Y	Y	N
Guo	Y	N	N
iDedup	N	Y	N
Founadation	N	N	Y

plication scheme, at the same time it makes the backup of different VMs to be independent for better fault tolerance.

The following table shows the strength and weakness of some well-know deduplication systems:

2. CHALLENGES

2.1 Indexing

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

The index needs to have three important properties: 1) scale to high capacities, 2) achieve good indexing throughput, and 3) provide high duplicate detection rateâĂŤi.e., high deduplication efficiency. Table 1 demonstrates how these goals become very challenging for a Petascale system. Consider a typical virtualized cloud in which 10,000 VMs store 1 PB data, and the chunk size is 4 KB (for fine-granularity duplicate detection), indexing capacity will need to be at least 10,000 GB to support all 250 billion objects in the system. Such an index is impossible to maintain in memory. Storing it on disk, however, would greatly reduce query throughput. To support 1000 concurrent snapshot backup tasks, an aggregate throughput of 20 GB/sec, would require the index and the whole dedupe system for that matter to provide a query service throughput of at least 5,000 Kops/sec. Trying to scale to 1 PB by storing the index on disk would make it impossible to achieve this level of performance. Making the chunk size larger (e.g., 128 KB) would make deduplication far more coarse and severely reduce its efficiency.

Even we can distribute the index over the cloud machines, the resources that can be allocated to system services are limited to minimize the negative impact to user experiences. So it becomes obvious that efficient, scalable indexing is a hard problem. On top of all other indexing challenges, one must point out that chunk FPs are cryptographic hashes, randomly distributed in the index. Adjacent index entries share no locality and any kind of simple readahead scheme could not amortize the cost of storing index entries on disk.

2.2 Deletion

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

For simplicity, many previous works only investigated simple reference counting without considering reliability and recoverability. Reference counting, however, suffers from low reliability, since it is vulnerable to lost or repeated updates: when errors occur some chunks may be updated and some may not. Complicated transaction rollback logic is required to make reference counts consistent. Moreover, if a chunk becomes corrupted, it is important to know which files are using it so as to recover the lost segment by backing up the file again. Unfortunately, reference counting cannot provide such information. Finally, there is almost no way to verify if the reference count is correct or not in a large dynamic system. Our field feedback indicates that power outages and data corruption are really not that rare. In real deployments, where data integrity and recoverability directly affect product reputation, simple reference counting is unsatisfactory.

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

Mark-and-sweep is generally a better solution. During the mark phase, all snapshot metadata are traversed so as to mark the used chunks. In the sweep phase all chunks are swept and unmarked chunks are reclaimed. This approach is very resilient to errors: at any time the process can simply be restarted with no negative side effects. Scalability, however, is an issue. To deal with 1 PB snapshots data, which may be represented by 10,000 GB FP index, If we account for an average deduplication factor of 10 (i.e., each chunk is referenced by 10 different snapshots), the total size of snapshot metadata that need to be read during the mark phase will be 100 TB. This alone will take almost 2 hours on a 1000-machine cloud. Furthermore, marking the in-use bits for 250 billion entries is no easy task. There is no way to put the bit map in memory. One might want to mitigate the poor performance of mark-and-sweep by doing it less frequently. But in practice this is not a viable option: customers always want to keep the utilization of the system close to its capacity so that a longer history can be stored. With daily snapshots taking place, systems rarely have the luxury to postpone deletion operations for a long time. In our field deployment, deletion is done once a day. More than 2 hours in each run is too much. In a large production-oriented dedupe system reference management needs to be very reliable and have good recoverability. It should tolerate errors

and always ensure correctness. Although mark-and-sweep provides these properties, its performance is proportional to the capacity of the system, thus limiting its scalability.

2.3 Fault Tolerance

Because deduplication storage system artificially creates data dependency among different VM users, it increases the risk of data availability. In large scale cloud, node failures happen at daily basis, the loss of access to a shared chunk can affect many VMs whose snapshots share this chunk. Without any control of the scope of data sharing, we can only increase the level of replication to enhance availability, which would significantly balance out the benefits of deduplication.

3. PROTOTYPE DESIGN

Our architecture is built on the Aliyun platform which provides the largest public VM cloud in China. A typical VM cluster in our cloud environment consists of from hundreds to thousands of physical machines, each of which can host tens of xen-based[**?**] VMs.

Aliyun has built a hadoop-like platform, which includes several highly scalable cloud infrastructure services to support runing large scale VM cloud service:

- **DFS**: a distributed file system that is optimized for many large and sequential reads or appends.
- **MapReduce**: a distributed data processing framework supports Map-Reduce[4].
- **MemCache**: a distributed memory object caching system.

Our snapshot system and the virtual machine management service rely on these basic cloud services to be functional: DFS holds the responsibility of managing physical disk storage in the cloud, all data needed for VM services, such as virtual disk images used by runtime VMs, and snapshot data for backup purposes, reside in this distributed file system. In addition, our snapshot system places the index of CDS in MemCache for deduplication, and uses MapReduce to facilitate the offline data processing. All the above services can easily find their open-source counterparts, which shows the generality of our architecture and deduplication scheme.

User control VMs through the virtual machine management service. During VM creation, a user chooses a pre-configured VM image or a snapshot that contains an OS, then the VM management service copies the corresponding VM image to her VM as the OS disk. Data disks can be created and mounted onto the VM in the same way, either empty or from an existing snapshot. All these virtual disks are represented as virtual disk image files in our underline runtime VM storage system. To avoid network latency and congestion, our distributed file system place the primary replica of VM's image files at physical machine of VM instance. When user deletes her VM, all the runtime virtual disk images and their snapshot data are removed from DFS.

In each VM, the runtime I/O between virtual machine and its virtual disks is tunneled by the virtual device driver (called TapDisk[6] at Xen). This is also where our snapshot system is built in. Three major snapshot operations are supported:

- *Write snapshot*: save the current state of virtual disk as a snapshot. This is also where data deduplication would take place. During snapshot backup, concurrent disk write is logged to ensure a consistent snapshot version is captured.
- *Read snapshot*: restore the state of virtual disk from a snapshot.
- *Delete snapshot*: delete a snapshot and reclaim the disk space for unused data.

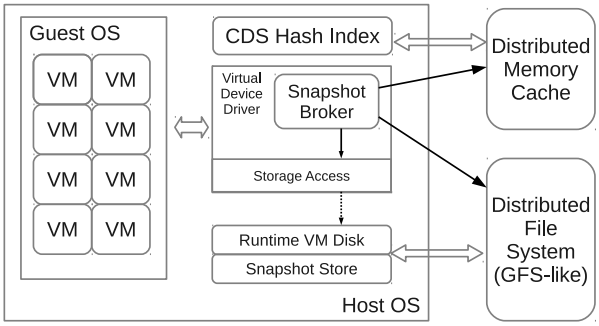

Figure 1: Snapshot backup architecture of each node.

Figure 1 shows the architecture view of our snapshot service at each node. The snapshot broker provides the functional interface for snapshot backup, access, and deletion. The inner-VM deduplication is conducted by the broker to access meta data in the snapshot data store and we discuss this in details in Section **??**. The cross-VM deduplication is conducted by the broker to access a common data set (CDS) (will discuss in Section **??**, whose block hash index is stored in a distributed memory cache.

3.1 Snapshot Deduplication

3.1.1 Snapshot Representation

The virtual device driver uses a bitmap to track the changes that have been made to virtual disk. Every bit in the bitmap represents a fix-sized (2MB) region called *segment*, indicates whether the segment is modified since last backup. Hence we could treat segment as the basic unit in snapshot backup similar to file in normal backup: a snapshot could share a segment with previous snapshot it is not changed. Moreover, we break segments into var-sized chunks (average 4KB) using content-based chunking algorithm, which brings the opportunity of fine-grained deduplication by allowing data sharing between segments.

As a result, the representation of each snapshot is designed as a two-level index data structure in the form of a hierarchical directed acyclic graph as shown in Figure 2. A snapshot

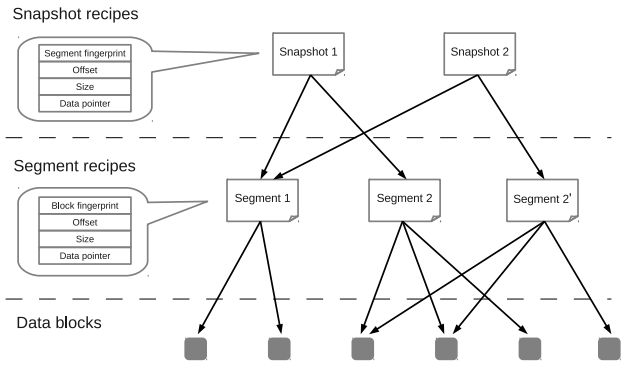

Figure 2: An example of snapshot representation.

recipe contains a list of segments, each of which is represented as a segment recipe that holds the meatdata of its chunks. We choose this two-level structure because in practice we observe that during each backup period only a small amount of VM data are added or modified. As the result, even the metadata of two snapshots can be highly similar, thus aggregating a large number of chunks as one segment can significantly reduce the space cost of snapshot metadata. Furthermore, instead of using variables-sized segments, we use a dirty bit to capture the change status of fix-sized segments which greatly ease the segment-level deduplication.

In both two kind of recipes, they do not include the actual data but only have references point to the data which are either stored in append store or CDS. In our implementation the data reference is a 8 bytes field which is either an ASID (discuss in 4 or an offset of An additional flag indicates

3.1.2 Multi-level Deduplication

The multi-level deduplication scheme is designed base on the observations on the VM snapshot data from production cloud. We found snapshot data duplication can be easily classified into two categories: *Inner-VM* and *Cross-VM*. Inner-VM duplication exists between VM's snapshots, because the majority of data are unchanged during each backup period. On the other hand, Cross-VM duplication is mainly due to widely-used software and libraries such as Linux and MySQL. As the result, different VMs tend to backup large amount of highly similar data. Our multi-level pipeline process can minimize the cost of deduplication while maximize the its efficiency at each level, and it is highly parallelizable since each segment is processed independently.

Inner-VM Deduplication. The first-level deduplication is logically localized within each VM. Such localization increases data independency between different VM backups, simplifies snapshot management and statistics collection, and facilitates parallel execution of snapshot operations.

Inner VM deduplication contains two levels of duplicate detection efforts:

- *Level 1 Segment modification detection.* If a segment is not changed, then its segment recipe can be simply reused by copying the data reference from previous snapshot recipe.
- *Level 2 Chunk fingerprint comparison.* If a segment is modified, we perform fine-grained deduplication by comparing the fingerprints of its chunks to the same segment's recipe in the previous snapshot, thus eliminate partial data duplication within the segment.

In general, operations at level 1 have almost no cost and most of unmodified data are filtered here. To process a dirty segment at level 2, there requires no more than one DFS access to load the segment recipe from previous snapshot, and a tiny amount of memory to hold it in main memory. *may need details here*

Cross-VM Deduplication.

3.2 Common Data Set

4. APPEND STORE

4.1 Introduction

Append Store (AS) is our underlining storage engine for storing snapshot data after deduplication. AS is built on top of our highly scalable distributed file system (DFS), which is very similar to Google's file system in a sense that it is optimized for large files and sequential read/append operations.

4.2 Design considerations

While scalability has been handled by the DFS, there are still several challenges in storing billions of var-sized small data chunks:

Locality perseveration Chunk data must be placed next to each other in the order of their writing sequence, because reading a snapshot incurs large sequential reads of chunks under their logical sequence in the snapshot.

Flexible referencing To avoid overwhelming metadata operation capbility of DFS's master, we have to group small chunks into large data files in DFS. However, snapshot deletion brings chunk data deletion, and reclaiming disk space would require many chunk data being moved. If such chunk movement incurs updates of data referencing in snapshot and segment recipes, then the cost of deleting a snapshot would be extremely high.

Efficient chunk lookup The translation from data reference to data location must be very efficient in two ways: first, a sorted index is necessary to fast locate the data location in $O(log(n)$ time; second, the size of index must be optimized to minimized its memory footprint and the cost of fetching index.

4.3 Architecture

Append Store supplies three interfaces: *get(ref)* accepts a data reference and retrives data, *put(data)* accepts data and returns a reference to be stored in metadata recipes, *delete(ref)* deletes the data pointed by the reference. Under the hood, small var-sized data are grouped and stored into larger data

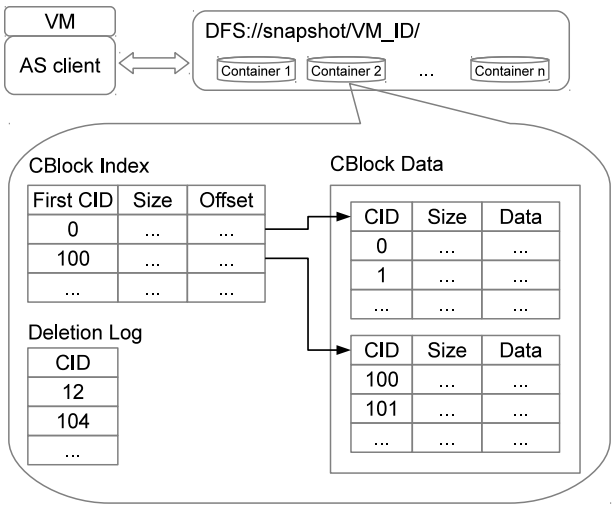

Figure 3: Architecture of Append Store

containers. Each VM has its snapshot data stored in its own Append Store, specified by the VM ID. We split every Append Store into multiple data containers so that reclaiming the disk space would not result in rewriting all the data at the same time.

As shown in Fig.3, every data container is represented as three data files in DFS: the data file holds all the actual data, the index file is responsible for translating data reference into data locations, and a deletion log file remembers all the deletion requests to the container.

A data reference is composed of two parts: a container ID (2 bytes) and CID (6 bytes). Append Store assign every piece of data a CID for its internal data referencing. When new data is appended, its CID is the current largest CID in that container plus one. As a result, all the data locations are naturally indexed by this self-incremental CID, no extra sorting is needed.

Append Store groups multiple chunk data (i.e., 100) into larger units, called *CBlock*. CBlock is the basic unit for append store's internal read/write/compression. There is one index entry in the container index corresponding to every CBlock. It keeps the first chunk's CID in that CBlock, and the CBlock data's size and location.

Using CBlock brings us several advantages: First, the write workload to DFS master is greatly reduced; second, grouping small chunks gives better compression. Third, reading a CBlock (200 - 600 KB) typically cost the same amount of disk seek as reading a 4KB chunk. Finally, this greatly reduces the size of index. Let m be the number of chunks in each CBlock, then the overall index size is reduced to $1/m$. In our implementation, using $m = 100$ reduces the index for a 1GB container from 10 MB to 100 KB.

4.4 Operations

In order to read a chunk data by reference, Append Store client first loads the container index file specified by the container ID, then search the CBlock index to find the entry that covers the chunk by CID. After that, it reads the whole CBlock data from DFS, decompress it, seek the exact chunk data specified by CID. Finally, the client updates itws internal chunk data cache with the newly loaded contents to anticipate future sequential reads.

Write requests to append store are accumlated. When the number reaches m, the AS client forms a CBlock by assigning every chunk a CID, compress the CBlock data, and append it to the CBlock data file. Then a new CBlock index entry is appended to CBlock index.

Append store adopts lazy delete strategy. The deletion requests are appended into every container's deletion log file with the CID of data to be deleted. CIDs in deletion log are guaranteed to be referenced by nobody and can be safe deleted in future. Periodically, snapshot management system asks append store to compact containers in order to reclaim disk space. The actual compaction will only take place when the number of deleted items reached $d\%$ of container's capacity. During compaction, append store creates a new container (with the same container ID) to replace the existing one. This is done by sequentially scan the old container, copying all the chunks that are not found in deletion log to the new container, creating new CBlocks and indices. However, every chunk's CID is plainly copied rather than re-generated. This does not affect the sorted order of CIDs in new container, but just leaving holes in CID values. As the result, all data references stored in upper level recipes are unaffected, and the data reading process is as efficient as before.

4.5 Analysis

Snapshot operation performance analysis:

The overall time of writing a snapshot is:

$$T_{write} = P_{dirty} * N_{seg} * [max(T_{scan}, T_{read\_AS}) + P_{change} * N_{chunk} * T_{check\_CDS} + P_{new} * N_{chunk} * T_{write\_AS}] \quad (1)$$

The overall time to read a snapshot is:

$$T_{read} = N_{seg} * T_{read\_AS} + N_{chunks} * [P_{in\_CDS} * T_{read_CDS} + (1 - P_{in\_CDS}) * T_{read\_chu} \quad (2)$$

Average time of reading a chunk:

$$T_{read\_chunk} = P_{in\_CDS} * T_{read\_CDS} + (1 - P_{in\_CDS}) * T_{read\_AS} \quad (3)$$

Time of reading append store:

Symbol	Description	Measurement
N_{seg}	Number of segments in the snapshot	
N_{chunk}	Number of chunks in one segment	
P_{dirty}	Probablity of a segment is dirty	
P_{new}	Percentage of new data	
P_{change}	Probablity that a chunk is not found in parent snapshot	
P_{in_CDS}	Percentage of chunks in CDS	
T_{scan}	Time to scan segment data and caculate content hash	74 ms
T_{write_AS}	Time of writing a chunk into append store	0.24 ms
T_{read_AS}	Time of reading a chunk from append store	
T_{read_chunk}	Time of reading a chunk	
T_{check_CDS}	Time of checking CDS	
T_{read_CDS}	Time of reading a chunk data from CDS data store	

Table 1: List of performance factors

$$T_{read\_AS} = 0 * P_{in\_cache} + (1 - P_{in\_cache}) * T_{load\_CBlock} \quad (4)$$

5. DELETION

In a mature VM cluster, snapshot deletions are as frequent as snapshot creations. Our system adopts lazy delete strategy so that all snapshot deletions are scheduled in the backup time window at midnight. Therefore, snapshot deletions must be fast enough to fit in time window and efficient enough to satisfy our resource constraints. However, there is no simple solution can achieve these goals with high reliability. In Aliyun's cloud, we use a *fuzzy deletion* method to trade deletion accuracy for speed and resource usage. This method tolerates a tiny percentage of storage leakage to make the deletion operation faster and efficient by an order of magnitude, yet we still adopt an slow but accurate deletion method to fix such leakage in the long-term. Our hybrid deletion strategy, using fuzzy deletion regularly and accurate deletion periodically, accomplishs our speed, resource usage and relibility goals very well.

5.1 Approximate Deletion

We design approximate deletion to fast identify unused data in the append store. Instead of scanning the whold append store, it uses a bloom filter to check if a data are still referenced by living snapshots. However, there is a small false-positive probility which would indetify unused data as in use, i.e., storage leakage. The following steps would take place during an approximate deletion:

1. **Creating bloom filter** Scan all the living snapshot recipess and their segment recipes, for every reference pointing to append store, add it to the bloom filter.

2. **Check existance** For every data reference in the deleted snapshot recipe and its segment recipes, check the existance of that data reference in bloom filter. If not found, it is safe to delete that piece of data from append store because no living snapshots has referenced it.

The overall time of running a approximate deletion for one snapshot deletion would be scanning all the living snapshots and deleted snapshots, since operations on the in-memory bloom filter can be done in parallel and is much faster than loading recipes from DFS:

$$T = (N_{SS} + 1) * T_{scan\_recipes} \quad (5)$$

Using the example and analysis in previous section, this approximate deletion can be done in 5 minutes. Memory usage of the bloom filter depends on its false-positive probility P_{bl}, when set P_{bl} to 0.01, the memory footprint of approximate deletion is about 15 MB.

The design goal of approximate deletion is to reduce the frequency of running accurate deletion. For one VM's snapshots, let $D_{leakage}$ be the amount of storage leakage, D_{del} be the average amount of data to be deleted in one snapshot deletion, then we have:

$$D_{leakage} = N_{approximate} * P_{bl} * D_{del} \quad (6)$$

where $N_{approximate}$ is the number of runs of approximate deletion. An accurate deletion is triggered to fix the storage leakage when $D_{leakage}/D_{del}$ is accmulated to exceed certain threshold T:

$$D_{leakage}/D_{del} = N_{approximate} * P_{bl} > T \Rightarrow N_{approximate} > T/P_{bl} \quad (7)$$

Therefore, when $P_{bl} = 0.01$ and $T = 1$, there would be a run of accurate deletion for every $T/P_{bl} = 100$ runs of approximate deletion. On a machine that hosts 20 VMs and each VM deletes snapshot daily, there would be less than one accurate deletion scheduled per day.

5.2 Accurate Deletion

Our accurate deletion uses mark-and-sweep to find all the unused chunks in a VM's append store. The following steps would take place during an accurate deletion:

1. Scan all the containers. For each container, extract full list of existing data references and allocate a bitmap. (Notice this list is self sorted.)

2. Scan all the snapshot recipes and segment recipes. For every data reference pointing to append store, find it in the above lists and mark the corresponding position in bitmaps.

3. Scan the bitmaps, for every bit that are not marked, tell append store the corresponding data reference can be deleted.

Although accurate deletion guarantees reliability and correctness, its speed and resource usage are big problems to our VM cloud. Let T_{scan_AS} be the time to scan the VM's append store, $T_{scan\_recipes}$ be the time to scan one snapshot recipe and its segment recipes, T_{mark} be the time of finding a data reference in the list and mark bitmap, N_{SS} be the number of snapshots and N_c be the number of chunks in one snapshot. The overall time of running an accurate deletion for one VM is:

$$T = T_{scan\_AS} + N_{SS} * T_{scan\_recipes} + N_{SS} * N_c * T_{mark} \quad (8)$$

For a virtual disk that has 10 snapshots, 50 GB of data in the append store, T_{scan_AS} would have cost half an hour already. In addition, scanning recipes of 10 snapshots cost another 5 minutes. Furthermore, maintaining the full lists of data references and bitmaps costs about 100 MB of memory. Having tens of VMs co-lived in one physical machine and deleting snapshot on a daily basis, it's infeasible to run accurate deletion as frequent as snapshot deletions.

5.3 Discussion

6. EXPERIMENTS

7. CONCLUSION

8. REFERENCES

[1] Aliyun Inc. http://www.aliyun.com.

[2] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[3] A. T. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized deduplication in SAN cluster file systems. page 8, June 2009.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Operating Systems Design and Implementation (OSDI âĂŹ*, 04, 2004.

[5] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. In *FAST '02: Proceedings of the Conference on File and Storage Technologies*, pages 89–101, Berkeley, CA, USA, 2002. USENIX Association.

[6] A. Warfield, S. Hand, K. Fraser, and T. Deegan. Facilitating the development of soft devices. page 22, Apr. 2005.

[7] B. Zhu, K. Li, and H. Patterson. 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*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.