GSnap: An Efficient Group-based Snapshot Index in the Cloud Database

XXX

ABSTRACT

Database often takes full and incremental snapshots of its fine-grained status, to support database recovery. In recent years, cloud native database and data warehouse tend to query those snapshots using time travel to do instant recovery, flashback, and data tracking. This requires the system to efficiently query large scale data blocks across continues snapshots. However, existing snapshot technologies do not meet the requirement. The write amplification of Copy-on- Write (CoW) introduces additional expensive I/O operations, which severely degrade performance. In the Redirect-on-Write (RoW) which is more widely used in cloud storage, the updated data blocks are scattered among the snapshots, resulting in a dependency between the snapshots. The dependency is not friendly to the continuous query on the snapshots.

In this paper, we observed that the access of snapshots has the characteristics of locality and continuity. We therefore propose an efficient RoW-based snapshot index, called GSnap, which groups snapshot indexes based on cloud database workload and access behavior of snapshot. Specifically, we use two key techniques: Shared-Subtrees Indexing (SSI) and Group-Based Dividing (GBD), to perform split dependency of the snapshot index. In-group snapshot indexes reduce memory overhead by sharing subtrees. The snapshot index dependency chain is divided into groups and there is no dependency on snapshot indexes between groups. The index can directly locate data blocks instead of iterative traversal. At the same time, the design of the snapshot index deletion method is adapted to the snapshot sparse deletion model. We have implemented a prototype system in Ceph. Evaluation results with datasets demonstrate that, compared with state-of-the-art techniques, GSnap could effectively balance the overhead between index memory usage and recovery time.

KEYWORDS

cloud native database, snapshot, database recovery, time travel

1 INTRODUCTION

Databases perform full and incremental backup to ensure data reliability and durability, and provide point-in-time recovery (PITR) capability for protection against accidental deletion or writes. Modern database backup copies [6, 13] are not saved in a separate location from original data. Backup clones immutable snapshot into writable volume and live mounts to do instant recovery, which is called Cloud Data Management (CDM) [49].

Cloud native databases can fully take the advantages of cloud storage to provide hundreds of thousands snapshots. Part of the snapshots are stored in the Cloud Block Storage (CBS) whereas the rest snapshots are archived to the Cloud Object Storage (COS) to support more snapshots and reduce their cost. The snapshots in CBS can be instantly cloned and mounted while archived snapshots in COS normally need to be copied to CBS for recovery. The snapshot could even be taken once every 10 minutes in Google Cloud Persistent disk[8], and the number of snapshots per volume is up to 5000 in Rackspace Cloud Block Storage [12]. Cloud vendors support a single snapshot instantly restore and flashblack. For example, AWS EBS [43] provides Fast Snapshot Restore (FSR) to accelerate volume creation from the snapshot and eliminates the latency of I/O operations on a block when it is accessed for the first time. Azure Backup [4] improves restore performance with Instant Restore capability for VM snapshot. Moreover, PolarDB [11] and Aliyun RDS [1] provide similar instant recovery capacity and features like flashback.

Considerable number of snapshots leads to a new trend that databases always separate business and data track on multiple snapshots. For example, MongoDB supports row-level query on the queryable snapshots, which are similar to the read-only MongoDB instances. Data warehouse such as Snowflake [14], AnalyticDB MySQL [3] and Delta Lake [7] also provide time travel capability on continuous snapshots to access historical data at any point within a specified time period. Besides, the declarative extensions [39, 41, 42] to SQL are developed to specify and run multi-snapshot computations conveniently for auditing and other forms of fact checking. The new trend has a continuous access feature instead of a single snapshot, and is similarly sensitive to instant recovery latency or even real-time query.

However, existing snapshot technologies do not meet the requirements. For Copy-on-Write (CoW) [18, 33], it introduces write amplification because the data blocks are copied to the snapshot, then the new data is written to the data blocks on the device. It is unacceptable for cloud providers that the write amplification problem severely degrades performance [30]. Thus they prefer to Redirect-on-Write (RoW). For RoW, the updated data blocks are scattered while the unupdated data blocks point to the previous snapshots. Restoring a snapshot requires access to several snapshots. Thus more additional snapshots need to be accessed if restore continuous snapshots, which increases the recovery latency and degrades the query performance of the snapshots.

Snapshot recovery includes index and data block recovery, which are both stored in the remote storage system. The snapshot builds indexes and then loads data blocks. Index recovery includes loading index from the remote node and locating the address of the data block locally. In RoW, index recovery needs to load more indexes because of the dependency, and the number of hops in locating is linearly related to the number of snapshots. Large index size increases recovery latency and memory overhead, which challenge network bandwidth and computing of the storage system. Many works tend to build data block indexes [25, 34, 47] to accelerate the recovery process. They cache the mapping from data block numbers to physical block addresses, which accelerate locating data blocks. However, fragmentation inevitably becomes worse as the database transactions run. The recovery process is still affected by the number of snapshots. Amazon EBS snapshot [2] copies the

complete index of the previous snapshot before updating data blocks to cut off the dependency. In the continuous access scenes, although the number of indexes to be loaded is reduced, the redundancy between indexes increases the local memory overhead.

Based on the workload of the database and the snapshot access characters, we propose a simple but effective snapshot index structure, called GSnap, to balance recovery overhead and memory overhead. The snapshot dependency is divided into groups, with each index relying on the previous indexes in its group. By sharing the subtrees in a group, the data block address could be obtained directly and the index memory overhead is decreased. Besides, prefetching indexes in a group instead of all dependent indexes could significantly reduce the number of snapshots and the recovery time. But grouping snapshot seems to be a non-trivial challenge. First, how to cut off the dependency between snapshots and share subtrees in a group? Second, how to adjust the length of a group to adapt to the snapshot size and update frequency? GSnap has two key technologies, which are Shared-Subtrees Indexing (SSI) and Group-Based Dividing (GBD), to solve the issues above. SSI sets two types of indexes in a group. Specifically, The first snapshot index is a complete B+ tree, and the subsequent snapshots are partial. The complete tree no longer depends on the former indexes, while the partial tree nodes point to previous indexes in the group, effectively reducing redundancy. GBD divides the group according to the snapshot update ratio and the recent average continuous access length. The purpose is to make index size of group stable and efficient cache management.

Subsequently, we implement GSnap, with SSI and GBD technologies. GSnap can balance the time and space overhead in snapshot recovery, while we adjust the GSnap deletion according to the sparse snapshot deletion mechanism. We integrated GSnap index into Ceph RBD that supports taking snapshots at scale and continuous recovering from any snapshot with low latency. We employ workloads derived from the standard TPC-C benchmark to evaluate the performance GSnap index.

To sum up, we make the following contributions:

- We propose GSnap, a high-performance snapshot index that could directly locate data blocks and balance the overhead of recovery time and index memory consumption.
- We design a group-based snapshot deletion method to adapt to the sparse snapshot deletion mechanism, which could effectively speed up snapshot deletion.
- We implement GSnap based on Ceph RBD. Evaluation shows that GSnap gains significant performance improvement against state-of-the-art block storage systems.

The rest of this paper is organized as follows. Section 2 presents background. The motivation is introduced in Section 3. The design and implementation of GSnap are proposed in Section 4. The performance evaluation is shown in Section 5. Section 6 discusses the related work and Section 7 concludes the paper.

2 BACKGROUND

2.1 Cloud Block Storage

More and more tenants deploy applications and store data on the cloud now. Cloud Block Storage (CBS) [23, 48, 50] provides large-volume storage devices for Elastic Compute Service. As shown in

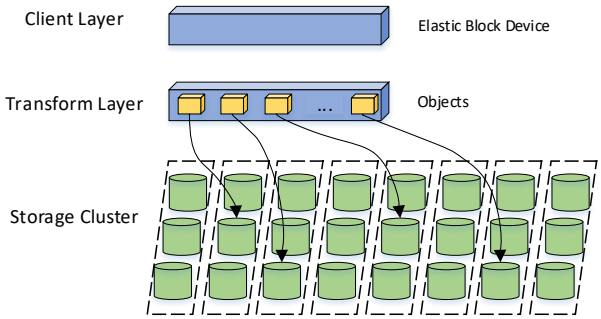


Figure 1: Architecture of Cloud Block Storage

Figure 1, CBS is made up of the client layer, data forwarding layer, and storage server layer. The client layer presents tenants with the view of elastic and isolated logic block devices allocated by the configuration and mounted to the client virtual machines. The block device is usually divided and striped into objects of the same size for storage and the object size can be configured in the storage server. The data forwarding layer maps data block physical location from the object number, and forwards I/O requests from the client to the storage server through a two-level hashing [26, 46], which spread replicas of an object to different nodes. Similarly, Openstack Swift [19] stores objects in partitions and transfer replicas of partitions to different regions by the ring, which is used to record the location information. The storage cluster layer is responsible for providing physical storage and typically employs replications to ensure data reliability and availability. Therefore, compared with the local block devices, the requests of CBS to access the data block inevitably pass through three layers to the storage cluster. Now Object Storage Service (OSS) [20] is also used for cloud storage, but it mainly stores unstructured data, such as documents, images, videos.

2.2 Snapshot Implementations

There are many classic snapshot technologies, such as CoW, CoW with background copy (CoW+), Split Mirror Redundant Array of Independent Disks (SMRAID), Continuous Data Protection (CDP) [21], and RoW. The different types of snapshot technologies have been extensively analyzed [17, 25, 29, 40]. In summary, the write overhead of CoW, CoW+, and SMRAID are suffered by their methodologies, their derivative technologies induce overheads for regular I/O and dramatic increase of sync operations when snapshots are present [27, 40]. CDP is mostly dependent on the underlying technology used for taking snapshots and network bandwidth.

For RoW-based snapshots, new data blocks write to the blocks belonging to the current snapshot, which stores the root node of the previous snapshot index for all old data blocks. RoW sacrifices contiguity in the original copy for lower overhead updates in the presence of snapshots. As the system runs and takes snapshots over a long time, the fragmentation of the data block becomes increasingly serious. When locating the data block, the snapshot checks whether it exists in the snapshot. If not, the snapshot looks for the earlier snapshots by the root nodes until the data block is located. Thus the process inevitably increases the number of hops. Iterative search degrades performance due to the low update

frequency of data blocks and the massive amount of snapshots. Snapshot taken by the Google Cloud Compute Engine Persistent Disk [8] only contains blocks that are different from the previous snapshot. Snapshots taken of the volume in Rackspace Cloud Block Storage [12] and virtual disk in Microsoft Azure [9] are incremental snapshots. Amazon EBS [43] has an optimized RoW snapshot index that cut off the index dependencies. It copies the entire previous snapshot index before updating the data blocks. Besides, the index uses a uniform region to store continuous unmodified data blocks to reduce the number of nodes. However, this index is not applicable in batch access mode since the indexes are loaded iteratively. Moreover, when the number of snapshots grows, the size of each region shrinks, and the number of regions grows, the index degenerates into a complete index, which increases memory overhead.

2.3 Vary Layers Supported Snapshot

Table 1: Summary of existing methods for the database backup and recovery (FS: file system, DB: database)

Methods	Mysqldump	Xtrabackup	BTRFS	LVM
layer	DB	DB	FS	Block
backup speed	slow	slow	fast	fast
restore speed	slow	medium	fast	fast

Snapshot technology can be implemented at different layers. Table 1 shows the existing methods supported by varying layers. In the database layer, administrator could choose mysqldump [10] for logical backup and Percona Xtrabackup [16] for physical backup. A logical backup stores the queries executed by transactions, while a physical backup copies the raw data to storage. In the recovery process, the stored queries are re-executed, or backup data are copied to a database directory. However, recovery operations in the existing tools take a long time, since these recovery procedures involve a large amount of I/O operations induced by database queries. Xtrabackup does not perform transactions and only copies original data, and then it is faster than mysqldump in restoring process, but the overhead still cannot be ignored.

Snapshots techniques supported by file systems and the block layer can be adopted to the backup and recovery of a database. Network Appliance NAS filers, the Sun ZFS filesystem [37] and BTRFS [36] organize the file as a tree with data at the leaf level and meta-data maintained in internal nodes. Each snapshot consists of a separate tree, but the snapshot may share subtrees with other snapshots. When the user produces a snapshot of the volume, simply duplicates the root node of the original volume as the new tree root node of the snapshot, and the new tree root is pointed to the same children as the original volume. Though creating a snapshot is light, the overhead of first write to a block is heavy because the entire tree of meta-data nodes needs to be copied and linked to the root node of the snapshot. LVM [22] operates between the file system and the storage device and provides fast snapshot creation and restoration using CoW. However, the CoW approach negatively affects run-time performance since it performs redundant writes due to data copies. We adopt tree structure to organize its data blocks like Parallax [31] and Ramdisk [32] to support rapid creation of snapshots.

2.4 Characters of Accessing Snapshots

In this subsection, we present our key snapshot observations about the behaviors of accessing. The characters can be exploited to greatly help design an optimal snapshot index that accelerates snapshot restoring and reduces index memory overhead.

As shown in Figure 2, we collect one-month snapshot information from AliCloud [5]. There were 115210 snapshots are preserved in the first two weeks, 522086 snapshots in the third week, and 634767 snapshots in the fourth week. Most of the snapshots saved in the last two week, because the system periodically takes snapshots at a fine density and filters snapshots based on the importance and date, which means continuously deleting snapshot sets of different lengths rather than batch deletion. Besides, earlier snapshots are easily deleted and merged. We also count snapshot accessed distribution, as shown in Figure 3. There are 2091 snapshots accessed within the first day, which accounted for 83.4% of the snapshots accessed throughout the month. Therefore, the feature of snapshot accessed is temporal locality.

Auditing and fact-checking [24, 38] are essential for applications now more than ever. In the multiple snapshot scenario, the user may retain the most accessed and recent snapshot set [44] to take unanticipated queries on long-term status. Besides, snapshots are usually loaded in deduplication systems as batches [35, 51]. The recovered snapshots are continuous and the length is indeterminate. Thus, the access behavior of snapshots is continuous. This behavior is proper for cloud tenants, who recover and mount more snapshots on different machines.

Existing RoW-based snapshot technologies suffer under continuous access workload. The reason is that the dependency between snapshots leads to loading all snapshots with reference relationships into memory instead of only snapshots accessed. The Amazon EBS snapshot index is a complete tree that stores all region's addresses to eliminate the dependency. However, it causes significant pressure on memory consumption because all indexes in memory are complete trees.

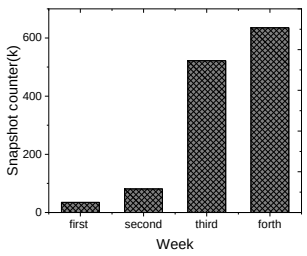


Figure 2: Snapshot retained distribution

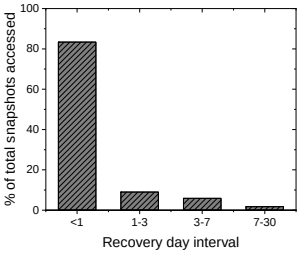


Figure 3: Snapshot accessed distribution

3 MOTIVATION

3.1 Fragmentation and Iterative Traversal

As discussed in Section 2, fragmentation causes serious iterative traversal in snapshot versions based on the RoW technique. In this subsection, we analyze the reason of iterative traversal with a typical example.

In traditional RoW, the root node points to the root node of the previous snapshot index, and new data blocks are written to

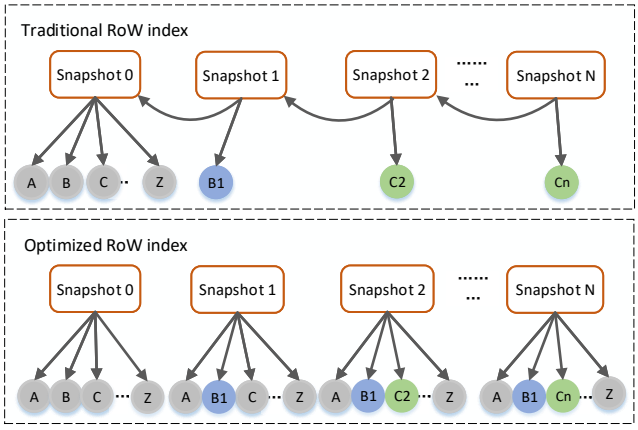


Figure 4: Example of unbounded traverses to restore snapshots on traditional versus optimized RoW index

the current snapshot. During the snapshot recovery phase, the recovery process requires access to other snapshots because the data blocks of the snapshot are scattered in other snapshots rather than only the current snapshot. The first step in locating an unmodified data block is to verify the destination snapshot, and then check the existence of the data block. As shown in Figure 4, there are *Block A*, *B* until to *Z*. *Block B* is updated in *snapshot 1*, *Block C* is updated in *snapshot 2*. When restoring *snapshot 1*, only *snapshot 0* and *snapshot 1* are verified. However, when restoring *snapshot N*, the recovery process first checks *Block A* in *snapshot N-1* while *Block A* is not updated in *snapshot N-1*. The recovery process has to access previous snapshots iteratively until found in *snapshot 1*. For *Block B*, the same traversal path is required. The recovery latency is affected by the data block fragmentation and the number of snapshots. For the optimized RoW, in Figure 4, the snapshot copies all snapshot numbers for unmodified data blocks at the end of the snapshot period. However, each snapshot index is a complete tree, which puts significant pressure on the memory overhead of the client node and snapshot loading from the storage cluster due to the snapshot access characteristics.

Amazon EBS [43] relieves memory pressure by replacing leaf nodes with regions based on the optimized RoW index. However, when the fragmentation of the data block increases, the size of the region is dropping while the number is increasing. Thus the index degenerates into the complete index which significantly boosts the memory usage. Under the snapshot feature of continuous accessing, EBS needs to iteratively load multiple snapshot indexes in distributed storage.

3.2 Optimal Snapshot Index

Under the observations and comparisons of existing snapshot indexes, fragmentation caused serious iterative traversal in snapshot versions based on the RoW technique. We present an optimal snapshot index namely GSnap, which cuts off the reference by putting the snapshot indexes into the group to eliminate iterative traversal.

An example of GSnap: For the *snapshot 1*, *2*, *3*, we put them into a group. *Snapshot 1* records the modified data block during the snapshot period, namely *Block B1*, and records all unmodified data blocks since it is the first snapshot in this group, thus its index is a complete tree. The *snapshot 2*, *3* no longer need to store addresses of the unmodified data blocks any more, and only store *Block C2* and *C3* respectively. When restoring a snapshot, we load the group into memory that means all snapshots in the group are restored. The snapshots after this group are divided into individual groups, which are organized similarly. The rationale behind this is to cut off the reference relationship between two groups while implementing incremental snapshots in a group. Only snapshots within the group need to be accessed to avoid iterative traversal when looking up the address of a data block.

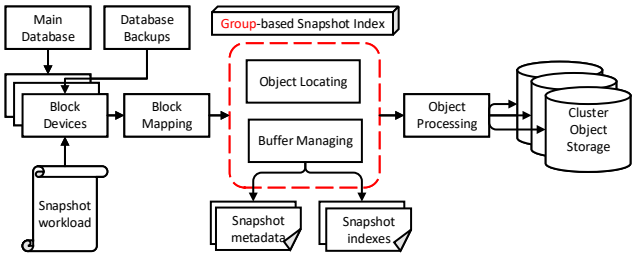


Figure 5: An overview of GSnap index framework

GSnap framework overview: GSnap manages the indexes within the group by using two key techniques. The overall workflow of the GSnap framework is shown in Figure 5, which includes three key stages: *Block Mapping*, *Indexing*, and *Object Processing*.

- **Shared-Subtree Indexing(SSI)**. The snapshot index is organized in the B+ tree structure and the leaf nodes store the addresses of the data blocks. SSI shares the internal and leaf nodes among the indexes. Hence, we only need to build and insert nodes when data blocks are modified to maintain that the path from the root node to the leaf node is complete, as detailed in Section 4.1.
- **Group-Based Dividing(GBD)**. GSnap triggers GBD once the data blocks updated in the group are saturated. GBD initializes the state of the new group to the active and flushes the achieved group to the storage cluster. Specifically, GSnap allocates the data block budget and sets the length of the group by Algorithm 1, as detailed in Section 4.2.

Block Mapping refers to the read/write requests of the database being mapped to the logical blocks of the block device through the Linux Storage Stack(LSS). The block device can be Rados Block Device(RBD) [45] or Network Block Device(NBD). Indexing accelerates the mapping of logical block numbers to physical blocks. Object Processing indicates that the subsequent requests are processed asynchronously via transactions after the network connection is established with the target object storage node. Write operation requests are duplicated across multiple nodes. In general, group-based snapshot index consists of two parts, data block locating and cache management. Data block locating works in snapshot accessing and group restoring. The cache loads snapshot indexes based on the access characteristics and adopts the LRU algorithm to swap out the snapshot group, which effectively reduces the index memory overhead.

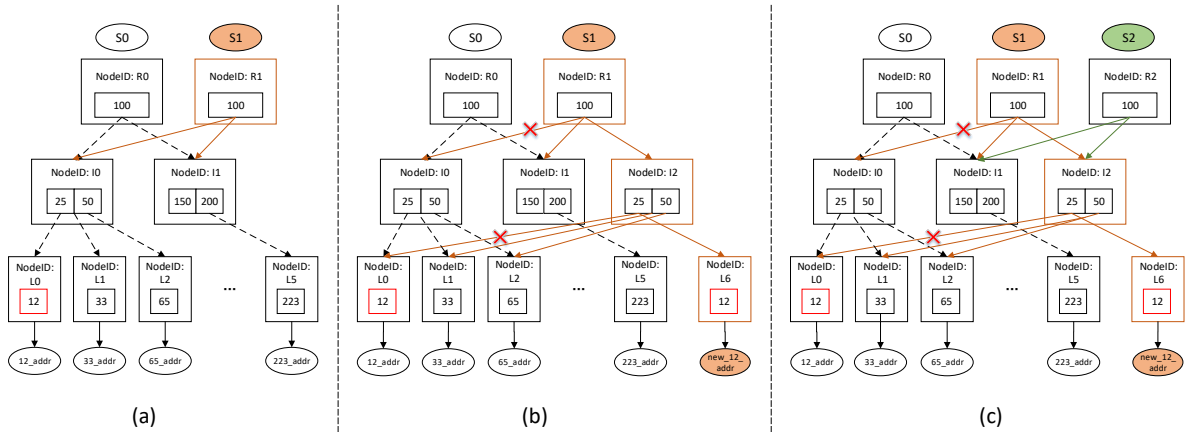


Figure 6: SnapTree: (a) Create S1 snapshot. (b) Block No.12 data write. (c) Create S2 snapshot

Challenges for GSnap. Actual snapshot workloads are much more complicated than the example shown in Figure 4, this results in inconsistent update frequency in each snapshot. Then the number of updated data blocks and the length of group are the key factors for taking and restoring snapshots. The update delay and memory usage of the index are proportional to the number of data blocks, we set the block budget for the latter group based on the block consumption of the former group to reduce memory overhead. The snapshot group length is still rewarded by the the former group so that ensure the indexes of each group occupy a close memory size for stable snapshot recovery and simple memory management. Besides, as we know from the previous section that users access snapshots continuously, we tend to set the length of the group is equal to the recent average access length of snapshots. We analyze those factors in detail in the next section.

4 DESIGN AND IMPLEMENTATION

4.1 Shared-Subtree Indexing

Snapshots also organize data blocks like the file system through a tree. The addresses of data blocks are stored in the leaf nodes. The root node and internal nodes point to the next level nodes, and the key in the node is the ID of the data block. Moreover, we set an ID namely *NodeID* for each node in the tree. The index just copies the root node of the last index as the new root node, which may share subtrees with other indexes once the user creates a snapshot. Data blocks shared with the snapshots can not be updated in place when write operations to a block device come. All writes must be redirected to new data blocks created by the current snapshot. The index creates leaf nodes for new data blocks, then checks whether the path from the leaf nodes to the root node is complete. If not, the index duplicates the internal nodes from the previous index, and finally modifies their pointers to redirect to the new leaf nodes. However, the nodes that record unmodified block numbers still point to the prior indexes.

We show SSI with an example in Figure 6 (a), when the user takes the snapshot *S1*, it just copies the root node from the snapshot tree *S0*, as the new root node namely *R1*, and *R1* points to the internal nodes *I0* and *I1* instead of pointing the root node *R0*. Figure 6 (b) shows that block *No.12*, whose address is recorded by the leaf node *L0*, is updated in the snapshot period. Specifically, *S1* creates a new leaf node *L6* and stores the address of the new block, then checks whether the path from the root node *R1* to the target leaf node *L6* is complete. If not, *S1* copies all internal nodes from *R1* to *L0*, and then changes the pointers to redirect to those new nodes. The root node *R1* points to internal node *I0* and changes to point to new internal node *I2*, then *I2* redirects to leaf node *L6* instead of *L0*. In the same way, the root node *R2* points to internal nodes *I1* and *I2* when taking the *S2* snapshot.

There are two advantages of the shared subtree structure. First, the subtree sharing among snapshots makes locating leaf nodes on any snaptree in the group with a constant time. The reason is that the number of hops to locate a leaf node is the same as the height of the tree regardless of the number of snapshots. Second, the memory footprint occupied by the tree is proportional to the number of updated data blocks in this snapshot instead of storing all nodes in each tree, and the index redundancy is reduced by only saving one copy of the address.

4.2 Group-Based Dividing

In this subsection, we introduce the Group-Based Dividing method, which is designed to cut off the snapshot index dependency and establish snapshot recovery granularity. There are two stages in GBD, the snapshotting stage and the dividing stage. In the snapshotting stage, the snapshot indexes generated in the group are partial trees. Besides, the state of the last index is *active* while others are *achieved*. In the dividing stage, the first snapshot index generated in the next group is a complete tree. The state transformation between the two stages is determined by the update ratio of the snapshots in the group and the latest average continuous access interval.

Snapshotting stage. In this stage, the system only copies the root node of the previous snapshot index for the new snapshot index. The new root node points to all internal nodes of the previous index and the number of snapshots in the group S plus 1. Besides, the *Created* flag of all internal nodes in the index are reset to false, which implies that no nodes are created in the index yet. As shown in Figure 7, when the data blocks updated dose not reach the saturation, the snapshot index S_1 only copies one root node, the reference of all leaf nodes in previous index is incremented by 1. At

the same time the previous index status is converted to *achieved*, and the current snapshot index status is *active*, which means all writes are updated in the current snapshot and only the current index could be modified during the period. We collect the update ratio of the former group P after the group state is changed. Once a data block is updated, the block counter O recorded in group increase by 1 and the *Created* flag of the new added internal nodes is set to true.

Dividing stage. When the number of snapshots L or the block counter O in the group reaches the threshold, the group state transforms into the dividing state. L and O are factors to trade off the recovery speed and memory overhead, and they will be discussed in Section 4.4. Figure 7 shows that the snapshot index S_n traverses all the leaf nodes of S_{n-1} and constructs a complete tree. Since traversing the leaf nodes only needs to access the root node of the last snapshot index and the number of hops is fixed, which speeds up the construction of the tree. The internal nodes updated in the last tree are marked. The system additionally copies the latest child nodes to the next group to reduce the delay in building a complete tree. GSnap allocates the block budget and set group length to the latter group based on the block consumption of the former group and the latest average continuous access snapshot length namely C. We constrain the continuously accessed snapshots in a group and adjust the size of the group based on the update ratio of snapshots. As shown in Algorithm 1, GSnap compares the block budget O^* with the block counter O and set the block and group length budget of the latter group. If the block counter O equals the block budget O^*, that means the block budget has been exhausted and the length of the group has not reached the threshold. GSnap rewards the latter group with the same update ratio and does not change the group length budget. The latter group is punished if the block budget is not used up, and the length budget is decreased to p, and p is the punish factor. In addition, we take the maximum value between L_{n+1} and the average continuous access length C to cover the snapshot set.

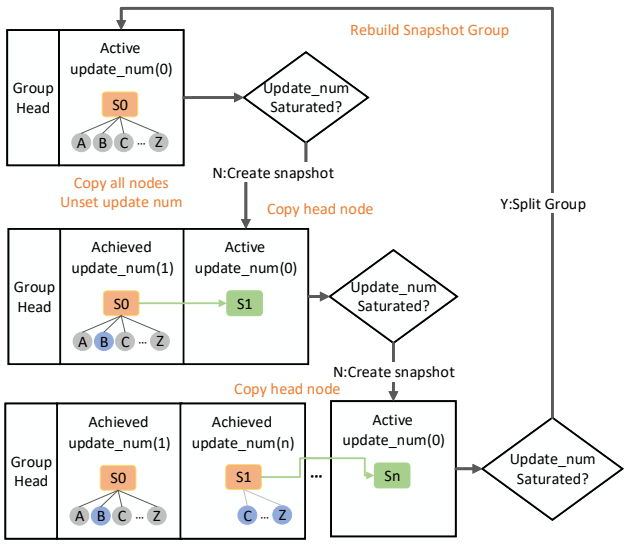


Figure 7: An example of GSnap index division

Algorithm 1: GSnap Index Group Dividing

Input: C - The latest average continuous access snapshot length; O_n^* - The object budget in the n-th group; L_n^* - The snapshot budget for the n-th group;

Output: SC_n^* - A set which stores pairs from snapshot to modified block count in the n-th group; O_n - The number of modified objects in the n-th group; L_n - The number of snapshots in the n-th group; P_n - The average update ratio of the n-th group; O_{n+1}^* - The object budget in the $(n+1)$-th group; L_{n+1}^* - The snapshot budget for the $(n+1)$-th group;

```
init SC_n, O_n, L_n, P_n;
while O*_n > O_n && S*.siz >= S_n do
    do taking snapshot and updating index;
    SC_n.insert(snapid, count);
    update O_n, L_n, P_n;
end
if O_n ≥ O*_n then
    L*_{n+1} = max(L_n,C); // length reward
else
    L*_{n+1} = max(p × L_n,C); // length punish
end
    split the group, and transform achieved group;
    O*_{n+1} = L*_{n+1} × P_n;
return L*_{n+1}, O*_{n+1};
```

4.3 Cache Management of GSnap

In this subsection, we introduce how to access the data blocks and cache management of the client host.

GSnap uses the shared memory to cache the snapshot index, which is appropriate for the interprocess communication. And the user could adjust the shared memory size to accommodate the snapshot size and access frequency by the profile. When mounting a block device, a daemon process is started in the system for data block locating and index cache management. As shown in Figure 8, the daemon process allocates and initializes the memory space, which include four partitions: ① *SM_Admin* includes *Cache_Allocter* and shared memory metadata, such as total space size, number of groups, etc. *Cache_Allocter* is responsible for the expansion of shared memory. The metadata is used to allocate memory segments to the group index and adopt LRU replacement algorithm to swap out the least recently accessed group index; ② *Snap_Admin* stores group metadata, and it is responsible for locating and loading the snapshot groups; ③ the snapshot index in the latest group from the storage node is loaded in *Latest_Group*, which refers to the snapshot index that has not yet formed a group; ④ the left space is used as a heap to store the indexes of groups.

When accessing the snapshot set, the daemon process checks whether the set has been loaded into memory. As shown in Algorithm 2, *Snap_Admin* compares the snapshot set NL with the group set AL that already loaded into memory. If *coverd* is true and NS is empty, target snapshot set indexes have been loaded into memory. NS contains all the snapshot segments which are needed to be loaded. Otherwise the daemon process traverses NS and *Group_Set* to determine the groups need to be loaded. Finally, *Snap_Admin*

finds the addresses of the index groups in *Group_Addr_Set*, and actively loads them from the cluster into the shared memory. We use *Group_Access_Counter* to count the number of accesses of the snapshot groups. Once the shared memory is not enough, we adopt a simple LRU algorithm to swap out the least recently accessed index group, and the granularity of swap out is the group instead of the snapshot.

Algorithm 2: Index Group Checking

Input: *AL* - A sorted set includes all pairs from group start to end snapshot id, and groups have been loaded into memory; *NL* - The pair of snapshots to be accessed, including the start to end snapshot id;
Output: *covered* - Whether to cover; *NS* - A linked list includes all snapshot segments need to load, including the start to end snapshot id;

```
init covered, NS, rear=GS.size;
if AL(0).first ≤ NL.first &&
  NL.second ≤ AL(rear).second then
    foreach AS in NL do
        if AS.first ≤ NL.first && AS.second ≥ NL.second
          then
            coverd = true;
            break;
        end
        if AL.first ≥ NL.first && AS.second ≤ NL.second
          then
            NS.append(NL.first, AS.first);
            NS.append(AS.second,NL.second);
        else
            NS.append(max(NL.second, AS.first),
              max(NL.first, AS.second));
        end
    end
else
    NS.append(uncovered set);
end
return coverd, NS;
```

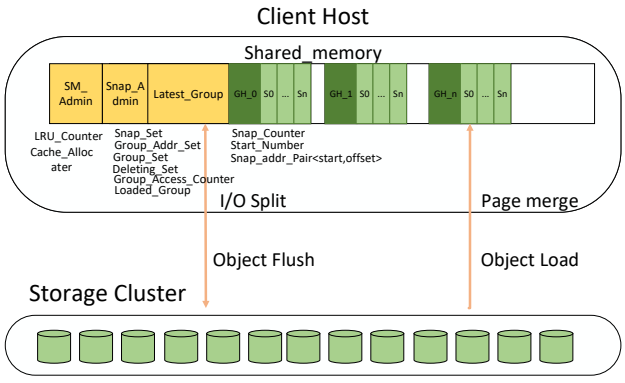


Figure 8: Cache management of GSnap index on client host

4.4 GSnap Recovery Model and Analysis

In this subsection, we theoretically analyze and compare GSnap with the traditional and optimized indexes, abbreviated as T-index and O-index, from two perspectives: the time cost of building an index and the memory overhead. We also determine the number of snapshots S and the object budget O based on the benefit.

When recovering and cloning a snapshot, the system loads snapshot group and constructs the index in memory first. However, due to the sharing of subtrees, the prerequisite for the construction of the current snapshot tree is that all the subtrees in the group have been constructed. The dependence chain length of the group is a key factor for snapshot recovery, to avoid loading indexes that do not be accessed. Further, the memory consumption of the group should be cautious. The reason is that the memory of the system allocated by the client node is limited. Therefore, we compare the existing indexes from these two perspectives respectively.

Table 2: Group recovery notation definition.

Notation	Description
D	The size of the block device (GB)
E	The object size of the block device (KB)
r	The longest snapshot interval with reference relationship
B	The network bandwidth of the storage cluster (Gbps)
T_a	The time cost of accessing memory, about 100 ns
T_r	The time cost of restoring a snapshot
M_r	The memory overhead of restoring a snapshot
M_c	The memory consumption of a compete tree
M_p	The memory consumption of a partial tree
T_t	The transmission time of each data block
T_i	The average time to insert a node in the snapshot index tree
T_l	The average time to locate a leaf node

Time overhead. Time overhead refers to the time span in which snapshot could provide services when accessing a certain snapshot. At the stage, the snapshot index is loaded into the memory and the data blocks are still in the physical cluster. We ignore the latency of the access request from the application layer. Even if the delay of the T-index and O-index is higher than the GSnap in loop access requests. We also limit the snapshot recovery to a group rather than across groups for simplicity. So we have the following equation to define the latency and memory overhead of restoring a snapshot. It may consist of three parts: ① transform the index blocks from the cluster to the client node; ② construct the tree and insert the addresses of all data blocks; ③ locate all leaf nodes when coping nodes to construct a complete tree from the former index. The respective latencies of the three stages are:

$$T_t = \frac{1}{32 \times B}$$
$$T_i > T_l = 8 \times \log_2(\frac{D}{E}) \times T_a \quad (1)$$

The O-index and GSnap contain traverse overhead because they may construct the complete tree. The time overhead of those indexes during the snapshot restoration is as follows:

$$
\begin{aligned}
T_r(T) &= \frac{D}{4} \times (r+1) \times T_t + 256 \times \frac{D}{E} \times p \times (r+1) \times T_i \\
T_r(O) &= \frac{D}{4} \times T_t + 256 \times \frac{D}{E} \times (T_i + T_l) \\
T_r(G) &= \frac{D}{4} \times (s) \times T_t + 256 \times \frac{D}{E} \times (1 + (s-1) \times p) \times \\
&\quad (n+1) \times T_i + 256 \times \frac{D}{E} \times T_l
\end{aligned} \tag{2}
$$

Assume $B = D$, the insert overhead of the tree is 4 times the traversal overhead. Then:

$$
\frac{T_r(T)}{T_r(G)} \leq \frac{16 \times (r+1) \times P}{16 \times S + 1} \tag{3}
$$

Memory overhead. Memory overhead refers to the memory consumption of constructing a snapshot index in the shared memory of the client host. We measure the memory overhead of each index. When the block device and the object size are fixed, the memory consumption of the complete snapshot index is also fixed. The memory consumption of the partial snapshot index is related to the update ratio. We simply set $M_p = pM_c$ MB. The memory overhead of different indexes during the snapshot restoration are as follows:

$$
\begin{aligned}
M_r(T) &= (r+1) \times M_p \\
M_r(O) &= M_c \\
M_r(G) &= M_c + (S-1) \times M_p
\end{aligned} \tag{4}
$$

Other snapshots with dependency are loaded in the T-index and GSnap, so we only compare the average memory consumption of each snapshot index. Obviously, compared with O-index, the memory overhead of GSnap is effectively reduced:

$$
\frac{M_r(O)}{M_r(G)} = \frac{S}{1 + (S-1) \times P} \tag{5}
$$

Balance load speed and memory consumption. Both snapshot index loading latency and memory consumption are tied to S and P. We comprehensively consider these parameters into loading speed and memory overhead and establish different weights in the system to satisfy the needs of snapshot recovery in various applications to explore the maximum benefit. We set the benefit as Δ, α and β as the weights respectively. The benefit function is the following:

$$
\Delta = \alpha \times \frac{S - (1 + (S-1) \times P)}{S} + \beta \times \frac{r + 1 - (S + \frac{1}{16 \times p})}{r+1} \tag{6}
$$

Assume $\alpha = \beta = 0.5$, simplify and conclude that: when $S = \sqrt{(1+r)(1-P)}$, the benefit Δ gets the maximum.

The above analysis shows that grouping snapshot index could effectively balance the recovery speed and memory usage, and determines the group size by workload parameters, namely r and P, to achieve the max benefit. However, in GSnap, to adapt to the snapshot continuous access, and the subsequent group length is also related to the snapshot update ratio and the average access length. The n-th group size is set to: $O_n = \max(S^*_{n-1}, S^*_n) \times P_{n-1}$.

4.5 Snapshot Deleting and Index Merging

To save the storage of the cluster, the system deletes historical snapshots with different lengths instead of simply deleting in batches. The system deletes snapshots based on their importance. Snapshots with low weight are more likely to be deleted. Earlier snapshots are less important. At the same time, snapshots taken at the critical timing are retained. This creates snapshots sets of varying degrees of sparseness.

Deleteing snapshot. Deleting a snapshot is divided into two parts. The first is to delete the data blocks contained in the snapshot, and the second is to modify the index of the snapshot. The data block of the snapshot is removed once it is not referred by other snapshots, that is, *Block_Reference* is equal to 0. Otherwise the reference is reduced by one. The system checks the group whether is in the *GS* to locate the address of the target index, then traverses all leaf nodes and decreases the number of updates *Current_Group_Update_Counter* for the current group by 1. When the traversal is completed, if the index has no child nodes, the group deletes the head node of the index, and decreases the number of snapshots in the group. Otherwise GSnap sets the index invalid in the group. Deleting the snapshot set needs to record the set to avoid an invalid merge operation. the thread tags the set in *Deleting_Set*, and clears the record when the snapshot set is deleted.

Merging index between groups. The index in the group is also removed after the snapshot is deleted. However, the memory assigned to this group is fixed, the group should be merged when the most of snapshots are deleted to reclaim the memory and improve snapshot recovery. For simplicity, snapshot merges are performed only between adjacent snapshot groups, which integrates the snapshot indexes of the latter group into the former group. So the condition of merging groups is that the free space of the former snapshot group could contain the latter group. In the latter group, only the first one is the complete snapshot tree, and the following snapshot indexes are partial snapshot trees. Thus the first complete snapshot tree in the latter group needs to be traversed, to rebuild as the partial snapshot tree in the former group. However, subsequent partial snapshot trees depend on the original complete snapshot tree, so all partial snapshot trees need to be regenerated in the first group. Specially, the merge process checks whether the groups to be merged are in *Deleting_Set*, and if so, it skips the merge operation and waits for the deletion. Then the process traverses all the leaf nodes of the partial snapshot tree in order, and regenerates the partial snapshot tree points to the snapshot tree in the former group.

4.6 Snapshot Interval Analysis of Merging Group

In the process of merging group, the leave nodes of all snapshots of the latter group are traversed, so the number of snapshots and the update ratio affect the merge efficiency. The benefit of merging groups is primarily accelerating snapshot recovery because only the required groups need to be loaded and no additional groups. At the same time, the memory utilization is improved. The number of partial snapshots in the latter group is the key factor when merging groups. Although the merge efficiency is improved if the number is tiny, there are a large number of internal fragments in the group before triggering the merge operation, which hinders the snapshot recovery and memory utilization. So we theoretically analyze the appropriate length of the latter group when merging.

Table 3: Group merging notation definition.

Notation	Description
k	The number of partial snapshots in the latter group
B_o	The average bytes of the nodes occupied by data block
T_n	the average node generation delay for one block
T_d	Disk I/O latency
T_n	Network I/O latency
B_d	Disk I/O block size
B_n	Network I/O block size

Merging overhead. Traversing the complete tree is required for the merging operation, while traversing the partial tree is optional. So the number of subsequent partial trees determines the latency of the merge operation. The merge overhead is determined by the number of partial indexes in the latter group, the average update ratio of snapshots, and the average node generation delay.

$$Overhead = k \times P \times T_n \tag{7}$$

Merging benefit. The benefit of merging operation consists of two parts. The first is to reclaim the memory space of the latter group, and the second is that only the target group which needs to be loaded without additional groups during snapshot recovery. Recycling the index could reduce the frequency of swapping out groups when the shared memory is fixed. We consider swapping out the group to disk as the main factor, and the first benefit of the merging operation is measured by the delay of swapping out the group. The recovery process loads at least one less group into the client node once the groups are merged from the storage cluster, and the second benefit of the merging operation is measured by the delay of loading group. We consider loading process only reduces one group and the delay is mainly network overhead.

$$Benefit = P \times L_c \times B_o \times (\frac{T_d}{B_d} + \frac{T_n}{B_n}) \tag{8}$$

Calculated by formulas 7 and 8, Merging snapshot group only benefits when $k <= \frac{L_c \times B_o \times (\frac{T_d}{B_d} + \frac{T_n}{Bn})}{T_n}$. So the merging of groups does not wait until only the complete snapshot tree remains and reduces the internal fragmentation of the group. Before the group merging, it first checks whether the counter of snapshots k in the latter group meets the condition, and then determine whether the space in the former group could accommodate the latter group. According to the experimental configuration in Section 5, the value of k is set $\frac{L_c}{5}$.

5 EVALUATION

We perform our experiments on a Ceph cluster with 4 cloud elastic compute services (ECS) running on Centos 7.9. Each ECS is equipped an Intel Xeon 2.5GHz processor with 32 cores, 32GB memory and 256GB SATA Disk and internal network bandwidth up to 10Gbps. The cluster configures a OSD node and a Monitor node on three hosts respectively, and the remaining host deploys a Ceph cluster client. In our evaluation, we built a GSnap index Embedded in Ceph v15.2.5. To comprehensively evaluate our design, we also embedded the T-index and O-index in the same version. We create block devices of different sizes through Ceph RBD interface, and then mount the block device on the client node. Besides we create the XFS file system on the block device, and deploy the database. We use MySQL 5.7 with InnoDB storage engine and running TPC-C benchmark. We use the TPC-C because it is a standard decision support benchmark with a well-known schema consisting of familiar tables. We run the all types of transactions in the benchmark. We report the Transactions Per Seconds (TPS) measured at the client node. For the MySQL/InnoDB configuration, we set the buffer pool as 1 GB and enable the direct I/O mode (O_DIRECT) to avoid the effect of file system page caching. All experimental results are the average value of 5 runs.

We create the Ceph client account on the client node as the administer and create block devices in a pool which supports three replicas by default. We configure block devices and their snapshots in the same pool to reduce the block migration. The size of the block device is 4GB by default and the punish factor p set as 3/4. Mysql is hosted on the XFS file system and default set to 4KB page size, which we did not change. We set the default disk I/O block size to 4KB and network I/O block size to 1MB.

5.1 Database Throughput

The system takes snapshot periodically in the initial phase before starting the TPCC [15] benchmark, and the time cost of creating a index is measured by the initial latency, which excludes the warm time of the benchmark and the time of the mount operation. Then we run the benchmark for 200 seconds in the snapshot period. For the convenience of evaluation, we do not delete snapshots.

To understand the performance of GSnap under database workloads, we evaluate the throughput on the various degrees of the connections with the other indexes. We do not perform snapshot recovery operations when creating snapshots, in order to reduce the shared memory usage. At the same time, when the snapshot group is completed, the group index is actively loaded into the remote storage cluster. We use the default script to create the database index and set the number of the database warehouses to 20. Specially, we adjust the connection of database to model the workloads.

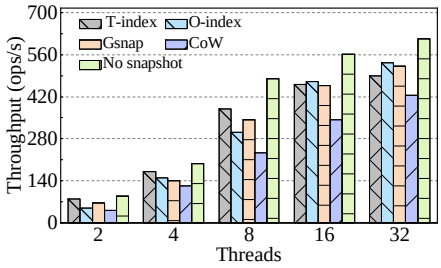


Figure 9: Throughput

As shown in Figure 9, GSnap outperforms the O-index for database workloads. The reason is that GSnap reduces the number of nodes, when the file system writes a new data block, Gsnap could directly look up the index to locate the target data block, while O-index needs to generate a node and then process it. But as the number of threads increases, the times of writing data blocks increases, and the nodes of the tree tend to be saturated, so O-index and GSnap are basically the same in index locating cost, so the

number of transactions they process is also close. GSnap is lower than No-snapshot by 9%. The reason is that the copy operation of the data block is reduced in RoW, but the snapshot still copies the uncovered data to the current object. The transaction throughput is affected by the database workload. We discuss the read/write ratio on snapshot performance in Section 5.7.

5.2 Snapshot Group Length

In this subsection we discuss the impact of snapshot group length on snapshot recovery. Since the snapshot update ratio and the average snapshot access length determine the length of the latter group, we fix the parameters and only measure the impact of the snapshot length on snapshot recovery. We create 100 snapshots to evaluate snapshot recovery and then randomly recovered a snapshot group. The benefits of GSnap compared to other indexes are highlighted by statistical recovery latency and memory overhead.

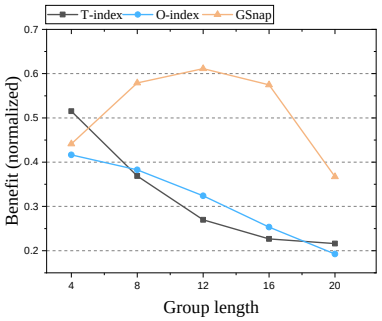


Figure 10: The benefit to restore a single snapshot group

As shown in Figure 10, GSnap cuts off snapshot references and shares child nodes within the group, which makes its revenue higher than both T-index and O-index. Because of the snapshot dependency, T-index loads its dependent snapshot indexes when restoring the snapshot collection, and in fact, the memory usage of the group is related to the length of the dependency chain. The performance of O-index degrades as the length of the snapshot set increases, because each index is a complete snapshot tree. GSnap has the largest profit when the snapshot group length is 12, which is close to the optimal value in the theoretical analysis. As the snapshot length increases, the number of snapshots that need to be loaded and the index memory increase, resulting in lower benefits, but higher than other indexes.

5.3 Snapshot Recovery

Before evaluating snapshot recovery, we clone all snapshots because snapshots are read-only and flush the group index out the shared memory on the client node, which ensures that the group index is loaded from the storage cluster when restoring the snapshot. The snapshots are not deleted when restoring process. There are two types of application scenarios for accessing snapshots: single snapshot and snapshot set. Therefore, we separately evaluate the recovery performance of those indexes in the two scenarios. We fix the length of snapshot group to 20 since the length of the snapshot group affect the snapshot recovery.

The recovery snapshot interval is set to 10, the purpose is to prevent the recovery of a single snapshot from evolving into the

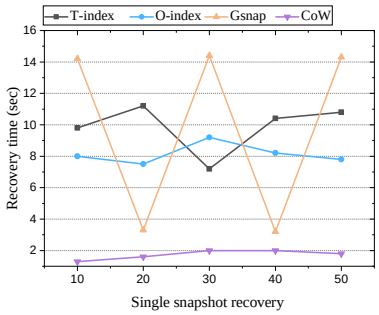


Figure 11: The latency to restore a single snapshot

recovery of a snapshot collection and the snapshots are recovered in order. As shown in Figure 11, we set up to restore the snapshots sequentially, so the snapshot indexes that the latter restored snapshots depend on have been loaded into shared memory. For GSnap, if the snapshot to be restored is already in the loaded group, the recovery latency is low, such as restoring the snapshots 20 and 40. But the recovery delay of GSnap exceeds O-index, the reason is that GSnap needs to load more indexes than O-index. All indexes in the group are loaded when GSnap restores a single snapshot, while O-index only needs to restore one snapshot index.

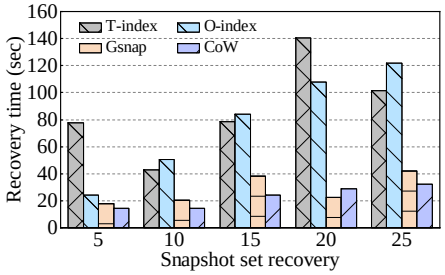


Figure 12: The throughput to restore snapshot set

We evaluate the delay of restoring snapshot sets of different lengths, to simulate continuous access to snapshots. We set all snapshot sets to randomly restore from the groups, and refresh the shared memory after each recovery operation to ensure the latency accuracy. As shown in Figure 12, compared with O-index, the speedup of GSnap is up to 79%, because in the loaded snapshot indexes, only 2 indexes are complete snapshot trees, and the others are partial snapshot trees, which can effectively reduce the index size of the transfer process. The recovery time of T-index deteriorates when restoring 20 snapshots, the reason is that the serial number of the first snapshot in recovery set is 34 and the dependency length of T-index is long.

5.4 Snapshot Deleting

In order to keep the number of snapshots constant, the system continuously deletes snapshots sparsely. We evaluate the latency of deleting snapshot sets of different lengths to keep the important snapshots. Besides, we also measure the overhead of merging the groups. Since CoW does not depend on previous snapshots, we only compare O-index, T-index and GSnap.

As shown in Figure 13, when GSnap deletes snapshots, the latency is close to T-index when the snapshot length is short, only

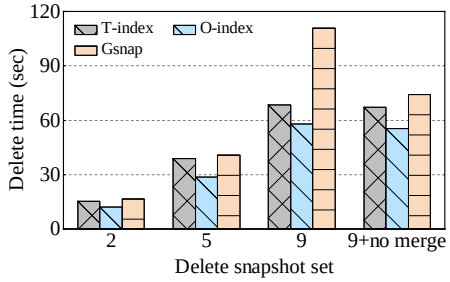


Figure 13: The latency to delete snapshot set

higher than 8%. This is because when GSnap deletes an index, it needs to traverse all leaf nodes to determine whether the data block can be deleted. We add an additional set of snapshot delete operation for the unmerged group, and merge operations degraded performance by only 26%. The reason is that 11 partial trees are left in the second group that triggered the merge operation, which increases the traversal overhead. The time of snapshot deletion is longer than GSnap due to the data merge and the index deletions operations. However, this cost makes the largest benefits to the performance of critical snapshot operations. First, the efficient snapshot creation operations make a small impact on its applications, thereby maintaining business continuity in a cloud computing environment. Second, the frequency of snapshot creation is much higher than the frequency of snapshot deletion.

5.5 Object Size

In this subsection, we evaluate the impact of different object sizes on index overhead and counter the proportion of the actual write size in the data block. Under the RoW mechanism, if the object is not fully written, the uncovered data needs to be copied from the previous snapshot to the current snapshot state. We set a balanced read-write ratio because this value affects the saturation of the snapshot index. We create 10 groups and randomly load snapshot groups, T-index and O-index load the corresponding snapshot sets. Finally we evaluate the memory consumption of the indexes.

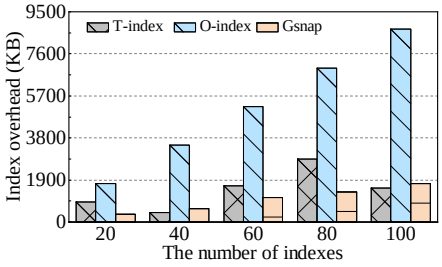


Figure 14: The index overhead of snapshot

As shown in Figure 14, the index overhead of O-index exceeds that of GSnap, even though the number of the object decreases as the size increases. The reason is that O-index is the complete snapshot tree, whereas in Gsnap, only the first index within the group is a complete snapshot tree, and the remaining indexes are partial snapshot trees. the design of the group dilutes the average number of nodes. The index overhead of GSnap is only 8% higher than that of T-index because the average indexing overhead is diluted by other partial indexes within the group. As the object size increases, the actual data write ratio decreases. The reason is that at the end of the snapshot, the system checks the coverage area of all objects and copies the default content from the previous objects to the current object.

5.6 Merge Length

In this subsection, we evaluate the effect of numbers of indexes in the latter group in the merge operation. Likewise, we set the length of the snapshot group to 20, and the recovered snapshot collection to 10. We randomly restore the set of snapshots to avoid trapping in a single group.

As shown in Figure 15, the merge overhead increases with the number of partial snapshots in the second group, because each index needs to be traversed and merged into the previous group. We evaluate load delay reduction of 28.7 seconds for one group, which is the benefit time of merging group. This delay is higher than the merge overhead when the number of the partial tree is 5 in the latter group, which means that the merge is only profitable at this time. The snapshot group length is related to the average access length, so the number of partial snapshots in the second group also dynamically adapts to user access behavior when merging groups. As shown in Figure 16, the more partial snapshots left when the merge operation is initiated in the second group, the lower the recovery latency. The reason is that most of the groups have been merged and the snapshots within the group are saturated, so only one group is restored. When there are 1 or 5 partial indexes in the group, merge operations are reduced. Thus the collection needs to restore two groups leading to increasing recovery overhead.

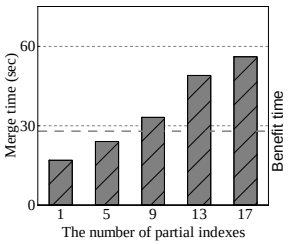


Figure 15: The latency of recovering snapshot set

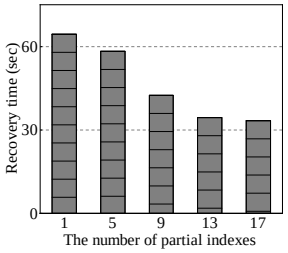
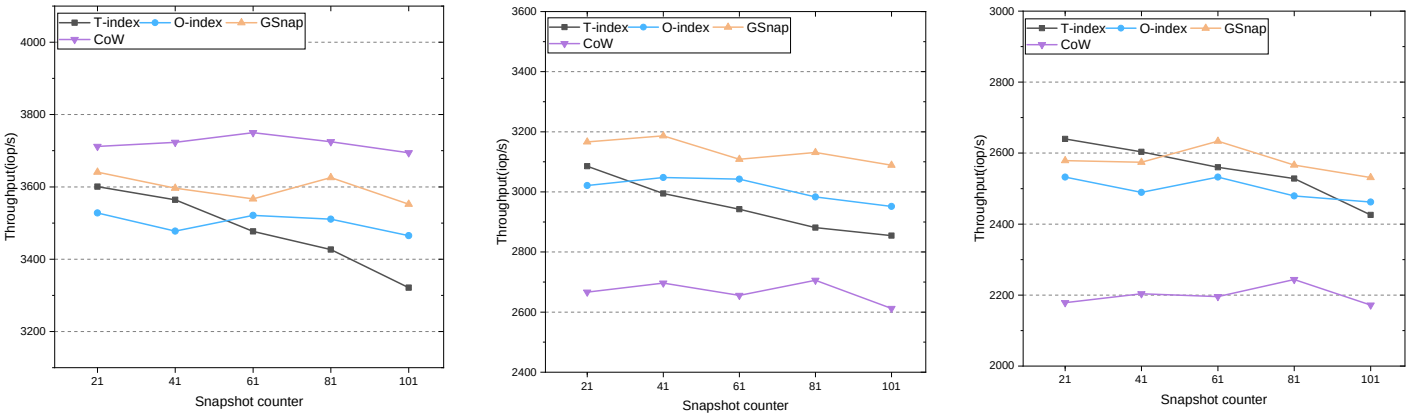


Figure 16: The latency of merging groups

5.7 I/O Performance

We also evaluate the I/O performance in the XFS and compare the throughput under various number of snapshot, highlighting the impact of indexes on locating data blocks. Specifically, the read/write ratio set as: read-only, balanced (50% read / 50% write) and write-only. the block size is default set to 4MB.

As shown in Figure 17, the performance of gsnap is stable as the number of snapshots increases. The reason is that the group cuts off the dependency of snapshots. When locating data blocks, only the snapshots in the group need to be accessed. Even if O-index only needs to access the previous snapshot, it copies all leaf nodes. Of course, the performance of the first snapshot in the group is lower than O-index since it accesses the prior group and constructs a complete index. In read-only workloads, CoW performs better than other indexes because only the current snapshot needs to be



(c) Read-Only (b) Balanced (a) Write-Only

Figure 17: The I/O performance for GSnap

accessed. As the write ratio increases, the performance degrades significantly due to the double write. The performance of T-index decreases as the number of snapshots increases in read-only workloads. Because it needs to access indexes iteratively when looking for data blocks. As shown in Figure 17(c), the iterative traversal of data blocks is reduced in the write-only workload, the performance is still degraded. The reason is that the lazy eviction of the previous indexes leads cache overhead increase.

6 RELATED WORK

Snapshot Technologies. Snapshot Implementations such as: CoW, CoW with background copy (CoW+), Split Mirror Redundant Array of Independent Disks (SMRAID), Continuous Data Protection (CDP) [21], RoW. The snapshot technologies have been extensively compared and analysed [17, 25, 29, 40]. CDP is made incrementally and used in synchronous data mirroring and Incremental Snapshot adapts a timestamp-based incremental approach with rollback provisioning. In summary, the write overhead of CoW, CoW+, SMRAID are suffered by their methodologies, their derivative technologies induce overheads for regular I/O and dramatic increase of sync operations when snapshots are present [27, 40]. Traversing to locate data blocks can cause performance degradation because fragmentation increases as snapshots increase. Amazon EBS [43] has an optimized RoW snapshot index that stores the regions which are continuous unchanged data blocks. The region points to the last changed snapshot instead of the previous snapshot to avoid iterative queries. However, when the number of snapshots grows, the size of each region shrinks and the number of regions grows, considerably increasing the memory footprint of indexes.

Index Structure. [28] uses the bitmap method to record the block number, but in the RoW, the dependency relationship still exists between the snapshots, and there is an iterative traversal to find the data block. [39, 41, 42] use a log structure to record snapshot metadata, and the metadata updating overhead of log structure is heavy when snapshots are continuously added and deleted. Parallax [31] leverages a tree structure to maintain snapshot indexes, and adopt ratix tree to reduce leaf nodes overhead. We also adopt the ratix tree structure and borrow the ZFS [37] to share child nodes between trees to reduce indexing overhead. Amazon EBS [43] optimizes the structure of the leaf node on the optimized index, setting fixed-length leaf nodes to store continuous unmodified data blocks to reduce the memory consumption of the index. However, the leaf node locating becomes complicated and the overhead of index updating increases when the snapshot is updated. Moreover, as the write ratio increases, the index degenerates the optimized index. Therefore, it is not applicable for the scenario of accessing a large number of snapshots.

Snapshot Implementation Layers. In the database layer, database administrators can choose mysqldump [10] for logical backup and Percona Xtrabackup [16] for physical backup. A logical backup stores the queries executed by transactions, while the physical backup copies the raw data to storage. In the recovery process, the stored queries are re-executed, or backup data are copied to a database directory. However, recovery operations in the existing tools take a long time, since these recovery procedures involve a large amount of I/O operations induced by database queries or raw data copies. Snapshots techniques supported by file systems and the block layer can be adopted for the backup and recovery of database. Network Appliance NAS filers, the Sun ZFS [37] and BTRFS [36] organize the file as a tree with data at the leaf level and meta-data maintained in internal nodes. LVM [22] operates the storage device and provides fast snapshot creation and restoration using CoW. However, the CoW negatively affects run-time performance since it performs redundant writes due to data copies.

7 CONCLUSIONS

In this paper, we propose GSnap, a lightweight RoW-based snapshot index for large-scale backup and fast recovery. It performs a hybrid design to split the dependency of snapshot indexes in a group. In-group snapshot indexes reduce memory overhead by sharing subtrees. The index can directly locate data blocks instead of iterative traversal. Loading target snapshot group indexes instead of indexes dependent, which balances index memory and recovery time overhead. The experimental results show that GSnap achieves higher performance than compared indexes.

REFERENCES

[1] [n.d.]. Aliyun RDS. https://www.alibabacloud.com/help/en/apsaradb-for-rds.
[2] [n.d.]. Amazon EBS Snapshot. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSSnapshots.html.
[3] [n.d.]. AnalyticDB MySQL. https://www.alibabacloud.com/product/analyticdb-for-mysql.
[4] [n.d.]. Azure Backup. https://learn.microsoft.com/en-us/azure/backup/backup-instant-restore-capability.
[5] [n.d.]. Backup setquery. https://www.alibabacloud.com/help/en/doc-detail/186041.html.
[6] [n.d.]. Cohesity backup-and-recovery. https://www.rubrik.com/insights/what-is-a-snapshot-backup.
[7] [n.d.]. Delta Lake. https://delta.io/.
[8] [n.d.]. Google Cloud Persistent Disk Snapshots. https://cloud.google.com/compute/docs/disks/snapshots.
[9] [n.d.]. Microsof tAzure VirtualDisk. https://learn.microsoft.com/en-us/azure/virtual-machines/disks-incremental-snapshots?tabs=azure-cli.
[10] [n.d.]. Mysqldump. http://dev.mysql.com/doc/refman/5.6/en/mysqldump.html..
[11] [n.d.]. Polardb. https://www.alibabacloud.com/product/polardb.
[12] [n.d.]. Rackspace Cloud Block Storage. https://docs.rackspace.com/support/how-to/prepare-your-cloud-block-storage-volume.
[13] [n.d.]. Rubrit snapshot backup. https://www.rubrik.com/insights/what-is-a-snapshot-backup.
[14] [n.d.]. snowflake. https://www.snowflake.com/en/.
[15] [n.d.]. Transaction Processing Performance Council. 2010. TPC-C: Decision Support Benchmark. https://www.tpc.org/tpcc/.
[16] [n.d.]. Xtrabackup. https://www.percona.com/software/mysql-database/percona-xtrabackup..
[17] Sameera Almulla, Youssef Iraqi, and Andrew Jones. 2013. A distributed snapshot framework for digital forensics evidence extraction and event reconstruction from cloud environment. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, Vol. 1. IEEE, 699–704.
[18] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2013. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 228–243.
[19] Swifts documentation. [n.d.]. In http://docs.openstack.org/developer/swift/.
[20] Michael Factor, Kalman Meth, Dalit Naor, Ohad Rodeh, and Julian Satran. 2005. Object storage: The future building block for storage systems. In *2005 IEEE International Symposium on Mass Storage Systems and Technology*. IEEE, 119–123.
[21] Neeta Garimella. 2006. Understanding and exploiting snapshot technology for data protection, Part 1: Snapshot technology overview. *IBM developerWorks, Apr* 26 (2006), 1–7.
[22] Michael Hasenstein. 2001. The logical volume manager (LVM). *White paper* (2001).
[23] Zhao Jin, Hanpin Wang, Lei Zhang, Bowen Zhang, Kun Gao, and Yongzhi Cao. 2019. Reasoning about Block-based Cloud Storage Systems. *arXiv preprint arXiv:1904.04442* (2019).
[24] Saehan Jo, Immanuel Trummer, Weicheng Yu, Xuezhi Wang, Cong Yu, Daniel Liu, and Niyati Mehta. 2019. Aggchecker: A fact-checking system for text summaries of relational data sets. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1938–1941.
[25] Linda Joseph and Rajeswari Mukesh. 2019. Securing and Self recovery of Virtual Machines in cloud with an Autonomic Approach using Snapshots. *Mobile Networks and Applications* 24, 4 (2019), 1240–1248.
[26] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 654–663.
[27] Jianxin Li, Hanqing Liu, Lei Cui, Bo Li, and Tianyu Wo. 2012. irow: An efficient live snapshot system for virtual machine disk. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*. IEEE, 376–383.
[28] Jianxin Li, Jingsheng Zheng, Lei Cui, and Renyu Yang. 2014. ConSnap: Taking continuous snapshots for running state protection of virtual machines. In *20th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2014, Hsinchu, Taiwan, December 16-19, 2014*. IEEE Computer Society, 677–684.
[29] S Mahipal and V Ceronmani Sharmila. 2021. Virtual Machine Security Problems and Countermeasures for Improving Quality of Service in Cloud Computing. In *2021 International Conference on Artificial Intelligence and Smart Systems (ICAIS)*. IEEE, 1319–1324.
[30] Alex Merenstein, Vasily Tarasov, Ali Anwar, Deepavali Bhagwat, Julie Lee, Lukas Rupprecht, Dimitris Skourtis, Yang Yang, and Erez Zadok. 2021. {CNSBench}: A Cloud Native Storage Benchmark. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 263–276.
[31] 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. ACM.
[32] Mark Nielsen. 1999. How to use a RAMdisk for Linux. *Linux Gazette (44)* (1999).
[33] Edmund B Nightingale, Kaushik Veeraraghavan, Peter M Chen, and Jason Flinn. 2008. Rethink the sync. *ACM Transactions on Computer Systems (TOCS)* 26, 3 (2008), 1–26.
[34] Irene Pekerskaya, Jian Pei, and Ke Wang. 2006. Mining changing regions from access-constrained snapshots: a cluster-embedded decision tree approach. *Journal of Intelligent Information Systems* 27, 3 (2006), 215–242.
[35] Yanjing Ren, Jingwei Li, Zuoru Yang, Patrick PC Lee, and Xiaosong Zhang. 2021. Accelerating Encrypted Deduplication via {SGX}. In *2021 {USENIX} Annual Technical Conference (USENIX ATC 21)*. 957–971.
[36] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)* 9, 3 (2013), 1–32.
[37] Ohad Rodeh and Avi Teperman. 2003. zFS-a scalable distributed file system using object disks. In *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings*. IEEE, 207–218.
[38] Ross Shaull, Liuba Shrira, and Hao Xu. 2008. Skippy: a new snapshot indexing method for time travel in the storage manager. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 637–648.
[39] Liuba Shrira and Hao Xu. 2005. Snap: Efficient snapshots for back-in-time execution. In *21st International Conference on Data Engineering (ICDE'05)*. IEEE, 434–445.
[40] Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2014. Snapshots in a Flash with ioSnap. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–14.
[41] Nikos Tsikoudis and Liuba Shrira. 2020. RID: Deduplicating Snapshot Computations. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2087–2101.
[42] Nikos Tsikoudis, Liuba Shrira, and Sara Cohen. 2018. RQL: Retrospective Computations over Snapshot Sets.. In *EDBT*. 600–611.
[43] Jinesh Varia and Sajee Mathew. 2014. Overview of amazon web services. *Amazon Web Services* 105 (2014).
[44] Michael Vrable, Stefan Savage, and Geoffrey M Voelker. 2009. Cumulus: Filesystem backup to the cloud. *ACM Transactions on Storage (TOS)* 5, 4 (2009), 1–28.
[45] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*. 307–320.
[46] Sage A Weil, Scott A Brandt, Ethan L Miller, and Carlos Maltzahn. 2006. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. IEEE, 31–31.
[47] Jing Yang, Qiang Cao, Xu Li, Changsheng Xie, and Qing Yang. 2011. ST-CDP: Snapshots in TRAP for continuous data protection. *IEEE Trans. Comput.* 61, 6 (2011), 753–766.
[48] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. 2020. OSCA: An online-model based cache allocation scheme in cloud block storage systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 785–798.
[49] Liang Zhao, Sherif Sakr, Anna Liu, and Athman Bouguettaya. 2014. *Cloud data management*. Springer.
[50] Ke Zhou, Yu Zhang, Ping Huang, Hua Wang, Yongguang Ji, Bin Cheng, and Ying Liu. 2018. Lea: A lazy eviction algorithm for ssd cache in cloud block storage. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 569–572.
[51] Xiangyu Zou, Jingsong Yuan, Philip Shilane, Wen Xia, Haijun Zhang, and Xuan Wang. 2021. The Dilemma between Deduplication and Locality: Can Both be Achieved?. In *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*. 171–185.