TiDedup: A New Distributed Deduplication Architecture for Ceph

Myoungwon Oh and Sungmin Lee, *Samsung Electronics Co.;* Samuel Just, *IBM;* Young Jin Yu and Duck-Ho Bae, *Samsung Electronics Co.;* Sage Weil, *Ceph Foundation;* Sangyeun Cho, *Samsung Electronics Co.;* Heon Y. Yeom, *Seoul National University*

https://www.usenix.org/conference/atc23/presentation/oh

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

TiDedup: A New Distributed Deduplication Architecture for Ceph

Myoungwon Oh[1] Sungmin Lee[1] Samuel Just[2] Young Jin Yu[1] Duck-Ho Bae[1]
Sage Weil[3] Sangyeun Cho[1] Heon Y. Yeom[4]
[1]*Samsung Electronics Co.* [2]*IBM* [3]*Ceph Foundation* [4]*Seoul National University*

Abstract

This paper presents *TiDedup*, a new cluster-level deduplication architecture for Ceph, a widely deployed distributed storage system. Ceph introduced a cluster-level deduplication design before; unfortunately, a few shortcomings have made it hard to use in production: (1) Deduplication of unique data incurs excessive metadata consumption; (2) Its serialized tiering mechanism has detrimental effects on foreground I/Os, and by design, only provides fixed-sized chunking algorithms; and (3) The existing reference count mechanism resorts to inefficient full scan of entire objects, and does not work with Ceph's snapshot. *TiDedup* effectively overcomes these shortcomings by introducing three novel schemes: Selective cluster-level crawling, an event-driven tiering mechanism with content defined chunking, and a reference correction method using a shared reference back pointer. We have fully validated *TiDedup* and integrated it into the Ceph mainline, ready for evaluation and deployment in various experimental and production environments. Our evaluation results show that *TiDedup* achieves up to 34% data reduction on real-world workloads, and when compared with the existing deduplication design, improves foreground I/O throughput by 50% during deduplication, and significantly reduces the scan time for reference correction by more than 50%.

1 Introduction

Open source infrastructure management systems [18,30] have helped cloud providers of different scales deliver services at low costs [4,7,14,37]. Progresses in corporate digitalization and new classes of data-intensive applications are fueling fast growth of data in the cloud and call for scalable and efficient data storage [17,34]. For such cloud serving storage, deduplication is expected to offer a means to mitigate the cost issue. However, most general-purpose distributed storage systems in use today do not provide cluster-level deduplication.

Several technical challenges partially explain why that is the case. First of all, all components in a successful deduplication architecture should be designed with scalability in mind. Moreover, the architecture should consider various data types, like file, block, and object, in search for data reduction opportunities. Last but not least, the architecture must have compatibility with existing services in the storage system, like snapshot operation. More comprehensive research is needed to make deduplication generally available in distributed storage systems (see Section 8 for further discussions).

The situation is no different in Ceph [42]. It has become a dominant open source distributed storage system in cloud environments thanks to its design that takes reliability and scalability as the first priority [7,31]. However, capacity optimizations like deduplication were not seriously considered in early designs, and thus, Ceph has failed to meet growing data demands in various installed configurations.

Recently, a cluster-level deduplication design was proposed for Ceph [29]. While the work presents worthy efforts (and is a baseline in our evaluation in Section 5), it contains critical technical issues that we believe will hold back its use in production environments. We capture three of them here.

First, the prior design blindly performs deduplication regardless of the amount of unique contents in a target object. Deduplicating a unique object brings no benefit and only increases computation and storage overheads. Since the amount of unique contents in objects depends highly on the workload, this approach might undermine deduplication efficiency, even more so in Ceph because it is a general purpose storage system for diverse data sets.

Second, the prior design relies on a limited tiering architecture that depends on coarse-grained processing and fixed size chunking (FSC). A single background thread of Object Storage Daemon (OSD) is responsible for all tasks required for deduplication. This architecture suffers performance degradation due to time-consuming object enumeration with a lock and a limitation on on-demand deduplication by external agents. Moreover, FSC is not always the most effective way to find duplicate chunks in real-world workloads [28,47].

Lastly, the prior design approach has drawbacks in its reference management method. Specifically, it does not work hand in hand with Ceph's snapshot feature because it lacks chunk reference management on an object snapshot. In addition, the reference counting method in the prior work takes a significant amount of time to identify reference mismatches on deduplicated chunks, because it requires full object search in the storage pool to count references.

In this work, we propose a new cluster-level deduplication architecture, called *TiDedup*, which does not have the aforementioned issues while respecting the core design principles of Ceph. Thanks to its design choices and implementation strategies, *TiDedup* effectively targets a general purpose distributed storage system for both primary and backup storage [10, 11], by providing file, object, and block services. *TiDedup* incorporates three key design schemes:

1. Selective cluster-level crawling. *TiDedup* implements a crawler process that incrementally searches and identifies redundant chunks to selectively trigger deduplication. By doing so, *TiDedup* successfully removes only redundant chunks with minimum overheads.

2. Event-driven tiering mechanism with content defined chunking (CDC). *TiDedup* eliminates background work in OSD and is designed to execute reactions upon an event to handle multiple requests concurrently. On top of that, new tiering APIs (*set_chunk*, *tier_flush*, *tier_evict*, and *tier_promote*), as well as control and data path with CDC, are introduced.

3. Object ID (OID) shared reference scheme. We propose an efficient reference management method using OID. By sharing OID reference information between adjacent snapshots, *TiDedup* not only makes deduplicated objects compatible with snapshot, but also minimizes messages between OSDs at snapshot creation time. In addition, by using OID as a backpointer, *scrub*—a job to identify and fix inconsistencies (e.g., reference mismatch)—is able to check whether the reference is valid without performing a full object search.

Our evaluation results show that *TiDedup* effectively saves storage space by up to 34% on real-world workloads including industrial manufacture data and corporate cloud services. Moreover, *TiDedup* outperforms the prior approach by 50% in throughput during deduplication. Importantly, our implementation passes the *teuthology test* [38], Ceph's quality test suite, demonstrating the robustness and readiness of *TiDedup* for real-world evaluation and production uses. This paper makes the following contributions:

• We demonstrate the challenges in modern distributed storage system when applying deduplication and overcome the challenges by introducing *TiDedup* (Section 3 and 4). Compared to the previous approach [29], *TiDedup* is scalable and compatible with Ceph's existing design philosophy and features. *TiDedup* allows the use of various chunking algorithms with extensibility, different from other tiering-based architectures [9, 49]. Our cluster-level reference management design is more efficient than existing reference count techniques [6].

• We propose a pluggable design that is applicable to Ceph and show how the design works in detail (Section 4). Since our proposal is based on a hash algorithm to locate objects, it can be applied to other systems that build on a similar hash algorithm, like GlusterFS [15], Swift [36] and Cassandra [19].

• We perform comprehensive evaluation using a realistic experimental setup (Section 5). Our evaluation shows that *TiDedup* effectively performs deduplication while having little impact on foreground I/O, and achieves more than 50% scrub time reduction, compared to the existing approach. We discuss design trade-offs based on evaluation results (Section 6).

• *TiDedup* has been merged into the Ceph main branch and is the default deduplication engine for Ceph. *TiDedup* is available to anyone for evaluation and production.

2 Background

This section provides a brief overview of Ceph and explains several key terms that will be used throughout this paper.

2.1 Ceph

Ceph [42] is an industry-leading open-source distributed storage system. It provides file, object, and block services to clients on a unified distributed object store called a Reliable Autonomic Distributed Object Store (RADOS), comprised of multiple OSDs. Ceph exploits a decentralized hash algorithm, known as CRUSH [43], to determine the object location within RADOS. An OSD is responsible for serving, replication, and recovery of objects on top of a block device. In this paper, we mainly deal with RADOS because it is the primary component involved with all three services.

Ceph uses three terms—POOL (tier), Placement Group (PG), and object name—to represent the object location. The tier is a global namespace that consists of PGs, and PG indicates a logical group of objects. In addition, every PG links a replication group, which maps a set of OSDs. By using Ceph's OID format—this includes the tier and object name—as an input argument of CRUSH calculation, Ceph can find out PG ID. Therefore, if an OID is given, Ceph is able to look up the OSD where the object is located. Note that objects that belong to different PGs can co-exist in the same OSD.

2.2 Deduplication

Deduplication is a well-known data reduction technique to eliminate redundancy in data [24, 44]. Its typical process is composed of the following three steps: chunking (e.g., fixed size chunking [33] and content-defined chunking [26, 46]), fingerprinting (e.g., sha256), and comparison with existing fingerprints [29, 46]. However, performing deduplication does not always lead to space savings because datasets have different amount of redundancy; for example, datasets of web and mail servers [25] may have lower duplicate data than backup [39] and registry workloads [16, 49].

2.3 How deduplication works with Ceph

Previously, Ceph proposed selective post processing deduplication based on a tiering mechanism [29]. In this design, Ceph divides a logical storage space into two groups: a base tier and a chunk tier. They manage metadata objects and chunk objects, respectively.

Upon a write request, every object is written to the base tier first as a metadata object; at this time, it is considered

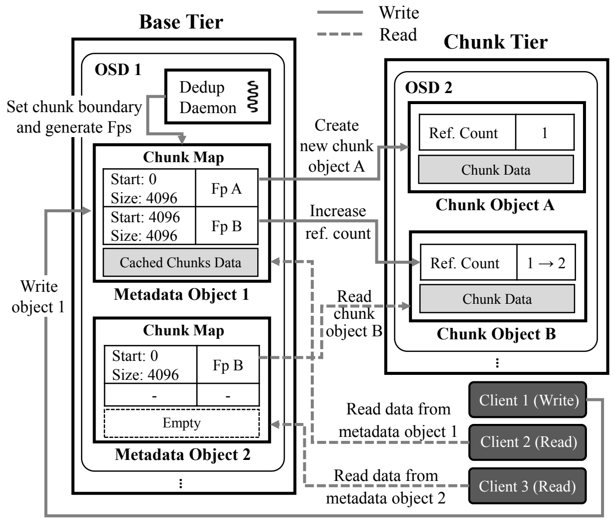


Figure 1: Ceph deduplication in prior work [29].

hot and is not eligible for deduplication. When the metadata object becomes cold (by LRU), OSD divides its content into several chunks using FSC, and generates a fingerprint from the corresponding chunk (e.g., Fp A and Fp B of Metadata Object 1 in Figure 1). Next, OSD stores the generated fingerprints in a chunk map of the metadata object. Then, using the fingerprint as OID, OSD stores the chunk as a chunk object in the chunk tier (Chunk Object A and B). If the chunk object already exists, OSD just increments the reference count by one. If not, OSD writes the chunk object, and sets the number of references to one.

To serve a read request, OSD searches metadata objects using the original OID. If the cached data resides in an object like Metadata Object 1 in Figure 1, the cached data is returned to Client 2. Otherwise, like Metadata Object 2, OSD retrieves the fingerprints from the chunk map, then it reads Chunk Object B by using the fingerprint (Fp B) as an OID. Finally, the OSD relays the data of Chunk Object B to Client 3.

Note that Ceph utilizes the CRUSH algorithm once again to calculate the chunk object location instead of maintaining a separate fingerprint index store. This technique is called double hashing [29].

2.4 Snapshot in Ceph

Ceph handles object snapshots via explicit APIs such as creation and deletion. OSD creates a new snapshot when a user makes a change on an object, at which point the version of the modified object is increased while keeping existing OID.

3 Motivation

3.1 Deduplication on unique chunks

In the prior approach, OSD decides whether an object needs to be deduplicated or not depending on which state the object is in between hot and cold instead of its redundancy; if the object is cold, OSD forces it to be deduplicated without checking its

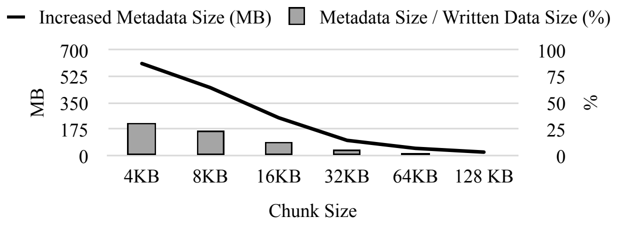


Figure 2: Metadata space overhead when deduplicating on unique objects. The total size of objects is 2 GB with 4 MB objects. Metadata size includes deduplication metadata (e.g., chunk information and fingerprint) and object (chunk and metadata) metadata.

redundancy. Therefore, the deduplication procedure inevitably generates unnecessary metadata even though the object has no redundant chunks. In the worst case, the amount of metadata could account for more than 30% of the total writes as shown in Figure 2. To eliminate the space overheads of the metadata, the best way is to remove only duplicate copies of chunks. To do so, however, we face the following two challenges:

Lack of knowledge of duplicate chunks. Ceph does not keep track of redundant objects. Moreover, to learn such information, each OSD would have to periodically check redundancy of all the objects in other OSDs, resulting in high overheads, interfering with foreground I/Os.

Object state and chunk-level management. The prior object management approach is unable to handle the cold object that has only unique chunks; when the object becomes cold, OSD deduplicates the cold object implicitly, degrading deduplication efficiency. To avoid this problem and maintain compatibility with the existing Ceph implementation, an additional state is needed to distinguish between unique and duplicate during the state transition from hot to cold. Moreover, there is a chance that an object has too few duplicate chunks to benefit from the reduced footprint of metadata. In this case, it's better not to deduplicate the object because the majority of the chunks are not redundant.

3.2 Structural limitations

Coarse-grained tiering mechanism. Ceph uses a coarse-grained tiering approach for deduplication; a background thread in each OSD updates the object state (hot or cold) and performs deduplication. However, it places a negative impact on the OSD foreground performance. In fact, to ensure consistency, Ceph holds a lock on PG even within an iterative loop for object enumeration to protect live objects in PG. Note that this PG lock is required when OSD handles a write operation to append a PG-level log for recovery. This is not a problem when the number of objects is small. However, if OSD contains a large number of objects, it is non-negligible because even simple object enumeration is very costly.

Moreover, the proposed tiering method does not expose the interfaces that allow other external modules to gather chunk information and trigger deduplication on demand. This strategy limits the effectiveness of deduplication.

No support for CDC. As an initial step, deduplication in Ceph was proposed based on FSC. FSC has the benefit of the low overhead calculation to determine chunk boundaries because it uses a predefined chunk size. However, it is well known that there is a boundary-shift problem [26] in some workloads. To avoid the problem, many studies have exploited CDC [10, 23, 24, 32]. We have tried complementing the fixed chunking's limitation using CDC. Unfortunately, applying CDC to the existing system leads to another challenge.

In the FSC approach, the user should configure the size of the chunk manually, then a background thread in OSD deduplicates the content of the chunk. This method is quite straightforward—there are only three operations: (1) set a chunk range (e.g., 8 KB chunk), (2) generate a fingerprint from the chunk, and 3) perform the content migration from the base tier to the chunk tier. In other words, once a range is set, the range remains valid until the range setting is changed. However, it does not seem well suited in terms of CDC because CDC generates different chunk sizes depending on the content; CDC may end up triggering multiple range changes even when a single chunk is mutated, so that the chunk ranges need to be recalculated.

Also, the prior approach uses three chunk states for deduplication: MISSING, DIRTY, and CLEAN. MISSING means that the content of the chunks does not exist in the base tier (not cached). DIRTY represents that the object was updated after the chunk was deduplicated, while CLEAN indicates that nothing has changed since the last deduplication operation. However, in CDC, there is no DIRTY state. If the chunk becomes dirty, its state becomes invalid as chunk range recalculation is required.

3.3 Inefficient reference management

The prior approach uses a false-positive based reference counting design [29]—while a chunk object is allowed to have a reference of deleted metadata object, the metadata object is guaranteed to always point to a valid chunk object—to prevent the failures. Nevertheless, it has other limitations, as follows:

Limitation of using the reference count. Reference mismatches may occur if OSD crashes and fails to successfully complete operations to update reference counts. To check reference mismatches in Ceph, the *scrub* process selects a chunk object, then scans all metadata objects to examine how many metadata objects have the reference of the chunk object; the expected reference count should be less than or equal to the reference count the chunk object holds. Note that the chunk object holds its own reference count, as explained in Section 2.3. Moreover, this behavior is repeated for all chunk objects, which in turn causes a scalability problem as the number of objects grows. The time complexity of the *scrub* process is $O(\#\text{chunk objects} \times \#\text{metadata objects})$.

Snapshot compatibility. In the previous approach, most of the features work with deduplicated metadata objects, but the snapshot feature does not, because reference management is missing for snapshot.

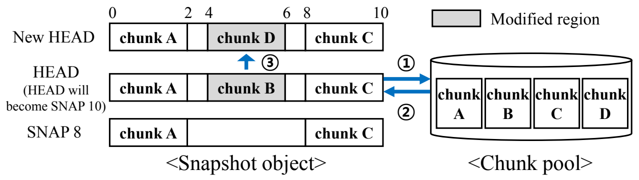


Figure 3: Ceph reference management (1. Send reference increment message, 2. ACK, 3. Update chunk information). SNAP represents a snapshot.

T	1	32	64	128	256	512
Latency (sec.)	0.13	0.32	0.72	1.8	3.96	9.37

Table 1: Snapshot creation time (T: the number of chunks on 4MB object, ten objects are used)

One straightforward yet naïve approach to supporting snapshot would be to increase (or decrease) the reference counts of all chunks of a deduplicated object whenever its snapshot is created (or deleted). However, this would generate an excessive number of messages among OSDs. Figure 3 describes the overhead. There is a HEAD that is the latest object version. For an incoming write to update the HEAD, OSD creates a snapshot (SNAP 10) in ascending order, excluding the new write. At this time, all chunk information within unmodified regions (chunks A and C) in the HEAD should be copied to the new HEAD's metadata ❸, after the reference increment (❶ and ❷) is done. Therefore, the snapshot creation will be delayed until OSD receives all ACKs for the reference increment. Note that an operation to increase reference count is synchronously done due to false-positive characteristics. Table 1 shows the snapshot creation time depending on the number of chunks. If there are 512 chunks on objects, the latency dramatically increases up to 9.37 seconds.

Aside from the delay, there is a redundant increase in the chunk's reference count if every snapshot has its own reference. For example, the reference count of chunks A and C will be three because new HEAD, SNAP 8 and SNAP 10 have the same chunks A and C in Figure 3. What is important is that even if the new HEAD, SNAP 8 and SNAP 10 are deleted, the reference count may not decrease due to the false-positive based reference counting strategy used. Although this reference leak can be fixed via the *scrub* process, it consumes additional space, until the *scrub* process is complete.

4 Design and Implementation of TiDedup

Figure 4 describes the overall structure of *TiDedup*. Basically, clients are not aware of the existence of the chunk tier and issue I/Os to the metadata objects in the same way as normal objects. For instance, the OSD stores content to the metadata object (W1) on a write request. In addition, upon a read request, OSD either returns the metadata object to the client immediately if it is cached (R1) or reads the content

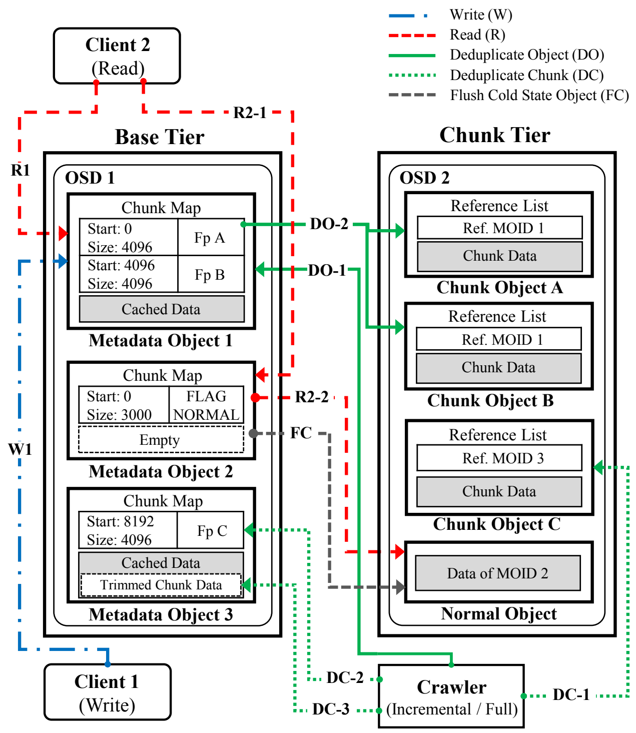


Figure 4: *TiDedup* architecture.

from the chunk object (R2-1 and R2-2). The crawler, which is responsible for triggering deduplication, scans objects—the scanning range is evenly divided and distributed to multiple threads in the crawler—on the base tier in either an incremental or full manner to look for redundant chunks. Then, the crawler checks the state of the metadata object using *stat*; *stat* retrieves the metadata object's information about hotness, deduplicated, and dirty. Depending on the state of the metadata object, the crawler works as follows.

• If the metadata object is hot, the crawler skips performing deduplication.

• If the metadata object is cold and contains more duplicate chunks than the threshold, the crawler performs deduplication if the object was not deduplicated before.

• If the metadata object is cold and has a few duplicate chunks, the crawler makes sure that the cached data is moved to the chunk tier (FC).

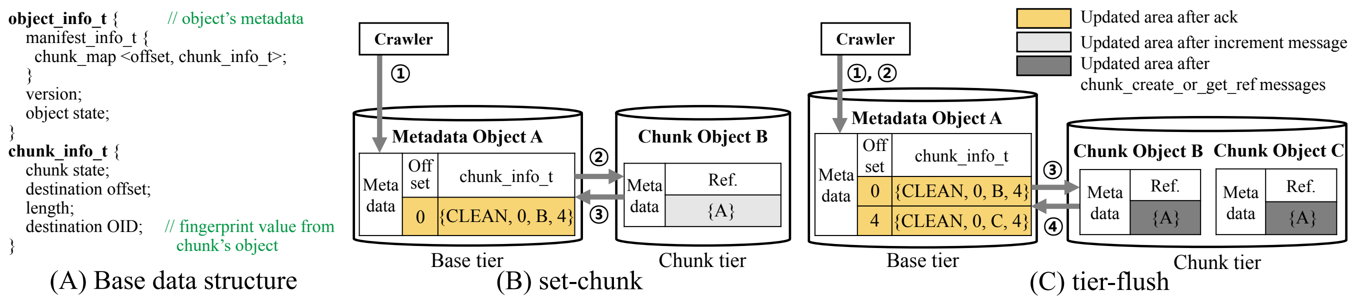
The crawler performs deduplication on the target object by using either *tier_flush* (DO1 and DO2 in Figure 4)—triggering CDC and moving chunks to chunk objects—or *set_chunk*—copying a target chunk to a chunk object in chunk tier (DC1) and setting a reference between metadata object and chunk object (DC2). Then, the crawler calls *tier_evict* to trim a range of the chunk in the metadata object, if necessary (DC3). These APIs will be described fully in Section 4.2.1.

4.1 Selective cluster-level crawling

To reduce the overhead as described in Section 3.1, we propose Selective cluster-level crawling, which crawls and carefully deduplicates objects on a base tier. The crawler is designed to run as a stand-alone application by decoupling a controller scheme—finding live objects and triggering deduplication—from OSD. Furthermore, we adopt incremental and full modes of crawling; both modes are introduced to figure out dedup-able objects on the base tier, but their crawling costs are different. In incremental mode, by choosing a small number of metadata objects gradually, the crawler reduces the chances of overutilizing system resources; during the daytime, it runs not to disturb user I/Os and unexpected high-priority I/Os, such as recovery and consistency checks. However, the incremental mode can not determine all dedup-able objects at once due to the limited range of search. To complement this weak point, full mode applies the full search without idle time. The drawback of the scheme is that it consumes more resources than the incremental mode does. Therefore, the crawler in full mode is scheduled to run once a week or at a longer interval, mostly during the nighttime when the system anticipates lower user I/O traffic.

In incremental mode, at first, the crawler gets a list of live objects sorted by OID on the base tier, then chooses a small group of objects from the list. Next, the crawler looks for redundant chunks among the selected objects—the crawler reads the objects, then runs CDC on the objects to calculate the fingerprint from the chunk, checking if the fingerprint is identical to other fingerprints collected before. The crawler considers the chunk a duplicate chunk if the chunk meets the condition; we define `chunk duplicate count`, which is the number of redundant chunks among the objects. If `chunk duplicate count` is higher than the threshold value K, the crawler regards the chunk as a good candidate for deduplication. Once duplicate chunks are found, the crawler has two tasks to do as follows. First, the crawler performs chunk-level deduplication (DC1, DC2, and DC3 in Figure 4). Second, the crawler checks information about how many duplicate chunks there are in an object. We call it `intra-object deduplication ratio`. The object, which has higher `intra-object deduplication ratio` than the threshold value L, is deduplicated via *tier_flush*, which is explained in the next section, by the crawler. The crawler pauses incremental mode when the tasks for the small group of objects are done, and resumes incremental mode, choosing the next small group of objects after the user-configured time T, 30 seconds by default. If there are no remaining objects in the list, the crawler refreshes a list of live objects. Then, it repeats the procedure in reversed direction while clearing the collected fingerprint list. In full mode, the differences from the incremental mode are the time interval and the search scope—it scans all objects in the base tier with a large time interval.

The crawler manages an in-memory fingerprint store to keep track of duplicate chunks; a key-value pair is used: $\langle$fingerprint : redundant count$\rangle$. The crawler updates the pair value when a fingerprint is calculated in either incremental



```
object_info_t {          // object's metadata
   manifest_info_t {
     chunk_map <offset, chunk_info_t>;
   }
   version;
   object state;
}
chunk_info_t {
   chunk state;
   destination offset;
   length;
   destination OID;    // fingerprint value from
                          chunk's object
}
```

Figure 5: Base data structure and two API procedures (*set_chunk* and *tier_flush*).

or full mode. If memory usage exceeds a given threshold value configured at startup, the crawler deletes all entries in the fingerprint store, except for duplicate fingerprints—the number of redundancies is higher than the threshold—information. This strategy might lose deduplication opportunities; the crawler is likely to drop fingerprints that appear once in a while under memory pressure. However, the crawler can become stateless, which overcomes practical issues via simple re-execution.

Object Management. A metadata object in a base tier can be in one of the following states: hot, cold, or deduped. When an object is newly created, its initial state is set to hot by default; the hot object is not deduplicated because it is more likely to be mutated in the near future. The transition from hot to cold takes place over time when *TiDedup* determines that a certain hot object needs to be evicted based on LRU. Then, the cold object is deduplicated if the corresponding object contains more duplicate chunks than `intra-object deduplication ratio` by the crawler. Otherwise, the cold object is migrated to the chunk tier without deduplication process (no chunking and fingerprinting).

TiDedup promotes chunk objects to the base tier if they are either updated or frequently accessed by read operations to avoid decoding overhead of deduplication; the object state is changed to a hot state at this time. Since we decide not to use the background thread at all, as explained in the following section, OSD keeps the object state using existing in-memory object metadata (*object_info_t*) and updates its state when the object is accessed.

4.2 Event-driven tiering mechanism with CDC

We design event-driven tiering to process multiple requests concurrently and minimize interference to foreground I/Os. Event-driven tiering does not perform any background works that can affect incoming I/O requests. Instead, it exposes APIs to the crawler to perform deduplication on demand. In addition, we redesign the overall I/O path to support CDC.

Basic read and write. Upon a read request, OSD looks up the metadata of the corresponding metadata object (*object_info_t*) where *manifest_info_t*—metadata related to deduplication—is stored. As shown in Figure 5 (A), *manifest_info_t* has *chunk_map*, which is a map data structure including source offset and *chunk_info_t*, to maintain mapping information between the source and destination chunk. Plus, each *chunk_info_t* has a state variable that indicates either MISSING or CLEAN. If the chunk state is MISSING, OSD calls *tier_promote* to move the chunk object from the chunk tier to the base tier in advance of responding to the user. If the chunk state is CLEAN, OSD replies to the user with the existing content the metadata object has.

For a write request, before storing content to the metadata object, OSD clears the chunk information (*chunk_info_t* in *chunk_map*) within modified range, while sending delete messages; if the write request overwrites a whole range of the object, no *chunk_info_t* exists. This is because modifying content means that the corresponding chunks are no longer meaningful—all chunk boundaries should be recalculated by CDC.

4.2.1 APIs with CDC

Set_chunk. The purpose of *set_chunk* is to set a chunk boundary within a metadata object for deduplication to support a case that only a specific range of the metadata object is redundant. To do so, *set_chunk* stores the given input argument—<source OID, destination OID, source version, source offset, length, and destination offset>—to the corresponding *chunk_info_t* in *chunk_map*. *Set_chunk* has two main roles: (1) to increase the target chunk' reference count, and (2) to update the chunk information.

As soon as OSD receives a *set_chunk* message from the crawler (❶ in Figure 5 (B)), it sends a message to add the reference of the source object ❷. Since calling *set_chunk* implies that the source object takes a reference of the target chunk object, reference increment should be conducted before the chunk information is stored. We will describe reference management in more detail in the next section. After the reference increment is completed ❸, *TiDedup* updates *chunk_info_t*. For example, when a user calls *set_chunk* with <source OID, fingerprint OID, v2, 4096, 8192, and 4096>, the OSD adds *chunk_info_t* (CLEAN, 4096, fingerprint OID, 8192) to *chunk_map* at offset 4096 if the object's version is v2; note that OID includes tier information where the object is located and OSD increases the object's version number whenever the object is changed. The crawler is required to maintain the target version to deduplicate objects correctly.

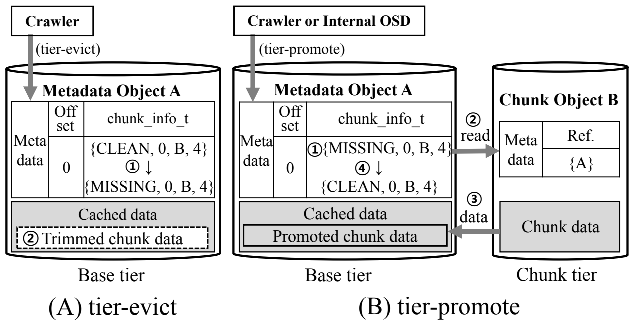


Figure 6: Procedure of *tier_evict* and *tier_promote*.

Set_chunk presumes that the target chunk is already copied to the destination OID before calling *set_chunk*, so the initial state of *chunk_info_t* is CLEAN.

Tier_flush. The crawler invokes *tier_flush* to remove redundancy on a metadata object, not a chunk; *tier_flush* deduplicates all contents on a metadata object explicitly if the object is not hot. Once *tier_flush* is called (❶ in Figure 5 (C)), OSD reads all contents from the metadata object, then executes CDC to generate chunks. After that, OSD generates a fingerprint value from each chunk ❷. Using given the fingerprint values as OID (B and C), OSD sends other OSDs in the chunk tier a compound operation, called *chunk_create_or_get_ref* ❸—this operation either increases the reference count if the target object is present or creates a new object with setting the reference count to one through transaction if the object does not exist. Once OSD receives all completion responses of the *chunk_create_or_get_ref*, it stores the metadata changes (e.g., *chunk_info_t*), and set the metadata object to deduped ❹. Although a failure can occur on rare occasions during *tier_flush*, *TiDedup* maintains consistency, because the metadata changes caused by *tier_flush* are not persistent until all the *chunk_create_or_get_ref* operations are completed. Note that *tier_flush* only updates metadata in terms of the metadata object. For example, if two out of ten *chunk_create_or_get_ref* are only successfully completed, *TiDedup* never updates the corresponding *chunk_info_t*. Instead, the crawler will retry to deduplicate the metadata object because it is not marked as deduped.

Tier_evict. *Tier_evict* removes the object's content using the punch-hole technique [20,22]. Figure 6 (A) demonstrates how *tier_evict* works. There is chunk B on the metadata object A. Once *tier_evict* is called, chunk B is marked as MISSING ❶, and then chunk B is trimmed from the metadata object A ❷, thereby resulting in a transition from a normal file to a sparse file. Note that the remaining parts of the metadata object A, which are not used as a chunk, remain on the metadata object A without removal. If a user tries to access the trimmed chunk, OSD in the base tier retrieves the original content from the chunk tier, before handling the user request.

Tier_promote. *Tier_promote* performs chunk migration from the chunk tier to the base tier even if a single chunk of metadata object is MISSING. Upon *tier_promote*, *TiDedup* finds chunks that have MISSING state in the metadata object ❶, as shown in Figure 6 (B), then sends read requests to corresponding chunk objects ❷. After the read requests are completed, *TiDedup* stores the given chunks in the base tier ❸. Unlike *tier_flush*, *tier_promote* updates chunk's content, and changes the chunk state from MISSING to CLEAN as soon as each read is succeeded ❹. For instance, although two out of ten reads are completed, *TiDedup* keeps the two completed chunks up-to-date.

4.3 OID shared reference management

To minimize *scrub* overhead and provide snapshot compatibility, we propose OID shared reference management design based on false-positive reference counting.

Data format for reference management. As described in the previous section, reference management using the number of references causes the *scrub* process to take a significant time to complete. To reduce the execution time for the *scrub* process, *TiDedup* makes use of Ceph's OID format to represent the reference instead of using a simple number. Since Ceph's OID contains location information, as described in Section 2.1, *TiDedup* can efficiently retrieve objects by their OID. In other words, if OID is used as a reference, *TiDedup* is able to recognize which metadata object refers to chunk objects like a back pointer and vice versa.

Scrub worker. *TiDedup* deploys a *scrub* worker as a separate thread in the crawler. *Scrub* worker wakes up periodically (the default value of wake-up period is equal to that in the full mode), then it begins to get a list of stored chunk objects on chunk tier and read their extended attributes one by one, each of which storing the reference information. Since the chunk object has its source OIDs, *scrub* worker does not need to read all metadata objects on the base tier to find reference mismatch. Instead, it just reads the metadata object whose OID is the source OID of the chunk object, then examines that the object has a reference to the chunk object. If the metadata object has a corresponding chunk object's OID, *scrub* worker repeats this until no more source OIDs that have not been checked present in the chunk object. If not, *scrub* process corrects the metadata object by removing the destination OID which is identical to the chunk object's OID.

Snapshot. To overcome snapshot-related limitations as described in Section 3.3, we introduce OID shared reference within an object. The key idea is that *TiDedup* does not generate an add/delete reference message if a chunk is identical—same offset, length, and destination OID—to a chunk in the adjacent snapshot. For instance, the reference count of BBB in Figure 7 is one; this means that only one add reference message is sent to chunk object BBB, because the chunk BBB in SNAP 7 is identical to chunk BBB in the new HEAD and SNAP 5, respectively. Note that we use the number of references here for the explanation.

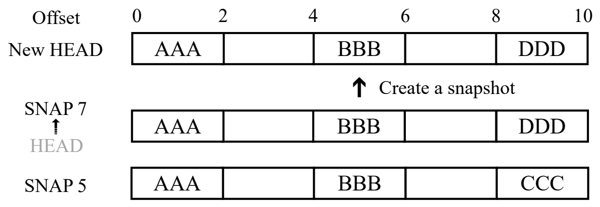


Figure 7: Snapshot reference management on snap creation.

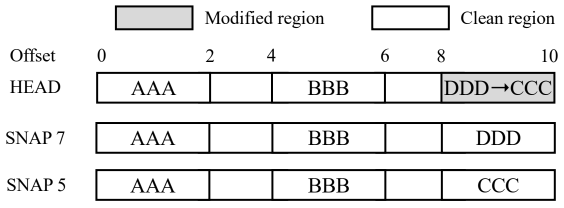


Figure 8: Snapshot reference management on overwrite (chunk DDD at offset 8 is changed to chunk CCC).

Figure 8 shows another example of a write operation. There is a modified region and clean regions in HEAD. *TiDedup* performs an add reference operation for chunk CCC—sending an add reference message then updating the corresponding *chunk_info_t* after the add reference message is done—, but does not generate a delete reference message for chunk DDD because SNAP 7 includes chunk DDD at offset 8. In addition, *TiDedup* does not do anything regarding all chunks in the clean region because each of the chunks is the same as the one in the previous snapshot.

However, a simple OID shared reference, as mentioned above, is insufficient to maintain consistency in reference management when the snapshot is removed. In Figure 8, SNAP 7 has a different chunk DDD compared to both SNAP 5 and HEAD at offset 8, so the reference count of chunk CCC is two, not one. At this point, if SNAP 7 is then removed, the reference count of chunk CCC will still remain two, even though it should be adjusted to one. To prevent this inconsistency, *TiDedup* checks both the prior snapshot and the next snapshot (SNAP 5 and HEAD in Figure 8) when the deletion occurs. If both chunks are identical, *TiDedup* sends a delete reference message.

As such, with OID shared reference, *TiDedup* generates only a limited number of add reference messages, regardless of many snapshot creations. Although *TiDedup* requires an additional search operation to identify the same chunks on the adjacent snapshots, this operation is relatively cheaper than handling the add reference operation.

OID shared reference can also work with proposed APIs, such as *set_chunk* (described in Section 4.2.1). In the *set_chunk* case; *TiDedup* exploits *set_chunk* to make a deduplicated chunk at any position in snapshots, as shown in Figure 9 (A), *TiDedup* performs an add operation for chunk AAA because there is no same chunk on the two snapshots (HEAD and SNAP 30), but nothing occurs in Figure 9 (B) because the HEAD includes chunk CCC. On the other hand, *TiDedup* generates a delete reference message for chunk CCC in Figure 9 (C) because both HEAD and SNAP 30 have the same chunk CCC.

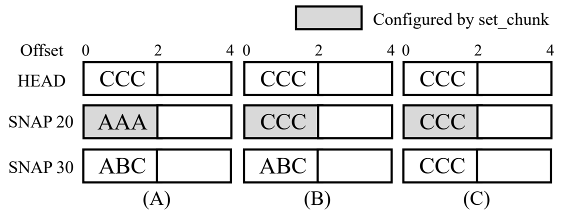


Figure 9: set_chunk (gray are new chunks by set_chunk).

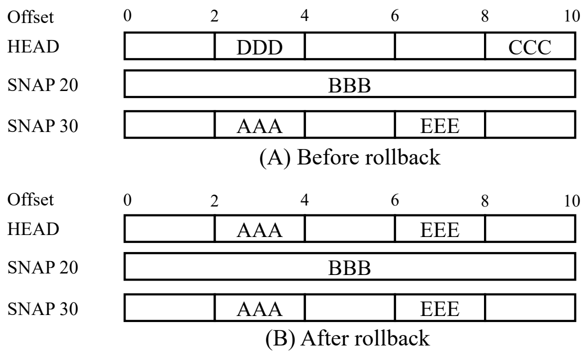


Figure 10: Rollback (HEAD is rollbacked to SNAP 30).

Rollback. *Rollback* replaces the HEAD with a given snapshot version. Upon *rollback*, *TiDedup* promotes all MISSING chunks both the HEAD and a given snapshot have from the chunk tier to the base tier. Then, the current HEAD is removed to make a correct clone into the HEAD. Figure 10 shows how shared reference count works with *rollback*. SNAP 30's chunks (chunks AAA and EEE) are copied to the HEAD (in Figure 10 (A)). Then, add operations for both AAA and EEE are needed because SNAP 20 has no identical chunks compared to the updated HEAD.

Crash consistency. OID shared reference management can maintain consistency after the crash because it is based on false-positive design. *TiDedup* guarantees the chunk object must exist if its metadata object has a reference. For this, the add operation for deduplicated chunks and update operation for relevant metadata are performed atomically. Moreover, *TiDedup* does not generate a delete reference message unless all shared chunk references in the object are unreferenced. Although *TiDedup* deletes *chunk_info_t* on sending a delete reference message without checking the completion of a delete operation—removing the metadata OID from the chunk object's extent attribute, this potential mismatch (a chunk object has a reference to a metadata object, but not vice versa) can be fixed by the *scrub* worker.

Reference set/get APIs. To set/get the reference, *TiDedup* adds four APIs that can be called by external clients and internal OSDs. *chunk_create_or_get_ref* creates a chunk object if the chunk object is not present, then adds a reference, which is the metadata OID. There are *reference_get* and *reference_put* to add/delete the reference of the chunk object. *read_reference* returns the list of references either the meta-

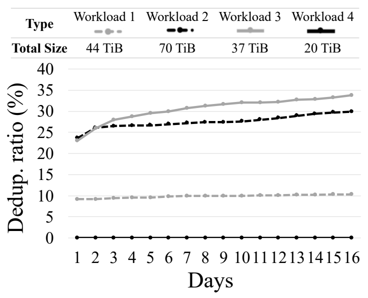


Figure 11: Space saving on factory data.

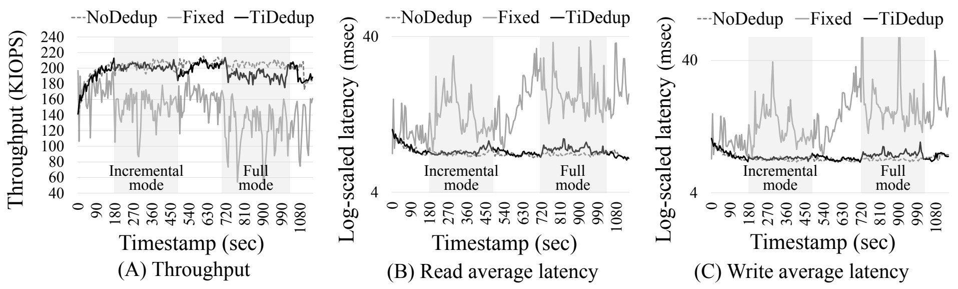


Figure 12: YCSB throughput (Workload a, record count is 500K, operation count is 20M).

data or the chunk object has. One important thing here is that the metadata object must return a reference list according to OID shared reference design; for example, in Figure 8, the reference list is {chunk AAA, BBB, CCC, CCC, and DDD}. On the other hand, the chunk object returns all references without considering OID shared reference.

5 Evaluation

5.1 Environmental setup

We use 10 machines in total, each of which is equipped with a 2-way AMD EPYC 7543 32-cores (80 threads) per NUMA node and 512 GB of DRAM. All nodes are connected with a 100 GbE network. Six nodes are used as storage nodes. Four nodes are clients to issue I/Os for evaluation. The latest version (Reef) of the Ceph storage cluster is configured by default parameters, except that the replication factor is two. Each machine has six BM1733 QLC SSDs (4 TB) and runs an OSD daemon on a single SSD. Note that we use high-performance equipments for evaluation to eliminate other performance factors. We use FastCDC [46] with the average chunk size set to 16KB. SHA1 is used to generate fingerprint value among the available fingerprint algorithm options (e.g., SHA1, SHA128, and SHA256). We set `Intra-object deduplication ratio` as 30% and `chunk duplication count` as four empirically. `NoDedup` represents default Ceph [42] without deduplication. `Fixed` means a prior deduplication approach for Ceph [29].

5.2 Deduplication ratio and throughput

5.2.1 Space saving

Internal cloud dataset. As shown in Table 2, `Fixed` shows limited data reduction compared to *TiDedup*. In `Logs` dataset, the data, which is partially mutated without data alignment, continues to be appended to Ceph cluster—the system monitor records similar logs from the cluster every 30 seconds, so that CDC is more effective than `Fixed`. In addition, CDC can save more space on `Virtual disks` dataset even if the dataset includes not only OS partitions but also user data. The reason is that the dataset is gathered from the internal developer cloud service, so user data includes duplicate data, such as mail, source code, and large images.

	Virtual disks			Logs		
Chunk size	8K	16K	32K	8K	16K	32K
Fixed	21%	12%	10%	5.7%	5.4%	5.3%
TiDedup	45%	36%	27%	18.5%	16%	12.6%

Table 2: Space saving on real-world datasets depends on the chunking algorithm and average chunk size. `Virtual disks` represents VMware vSphere images (10.1 TB) from a developer cloud service (67 users). `Logs` represents service logs (560 GB) for cloud infrastructure including monitoring and device state.

Factory dataset. We replay factory dataset generated during the semiconductor manufacturing on Ceph's object service (RGW). The factory dataset is normally used to detect or predict malfunctions of semiconductor products. We collect four types of data in total on a daily basis. As shown in Figure 11, *TiDedup* can achieve a high deduplication ratio (up to 30%) on Workload 2 (chip information during manufacturing). This is because Workload 2 has time-series monitoring logs having periodic values—the similar structured data is consistently appended. Unlike Workload 2, Workload 1 (equipment status) also includes time-series logs but has a smaller entry size—small tables with timestamps. This leads to less data reduction. Workload 3 (logs for photo lithography) has daily archive files stored in an incremental manner. It contains a large amount of redundant data and shows a high deduplication ratio. Workload 4 consists of metrology and inspection image files which are already compressed and contain little redundancy. Thus, it is not affected by deduplication at all.

5.2.2 Throughput

YCSB. We use YCSB workload a (read/write ratio is 50:50) to generate foreground I/Os. YCSB runs on four clients with sixteen threads using Ceph's block device service (RBD). As shown in Figure 12, the crawler is launched with incremental (at 180 seconds) and full mode (at 720 seconds), respectively—each of which runs for 300 seconds. Note that the elapsed time for incremental mode includes user idle time.

Figure 12 (A), (B), and (C) shows throughput and average latency. With *TiDedup*, throughput is not degraded significantly compared to `NoDedup` because *TiDedup* does not trigger the deduplication aggressively until the object is cold. *TiDedup* also achieves near-constant throughput unlike `Fixed`, which suffers a significant performance drop because a background thread blocks foreground OSD I/Os. Moreover, *TiD-*

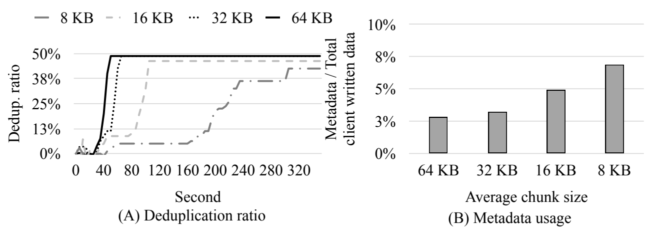


Figure 13: Space saving depending on the average chunk size.

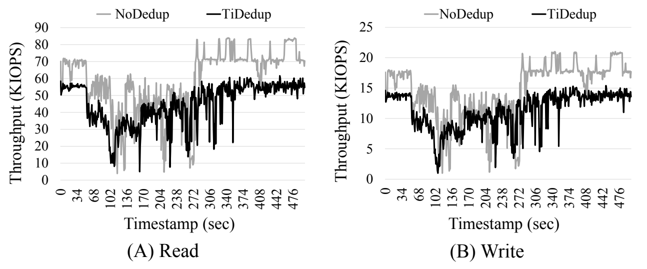


Figure 14: Recovery performance (Average IOPS of four clients).

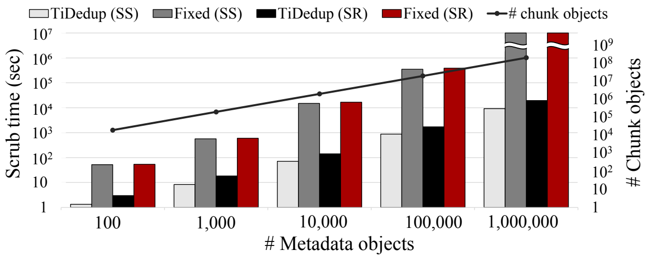


Figure 15: Scrub time comparison (SS: `Scrub-search`, SR: `Scrub-repair`). SR time includes SS time.

edup with incremental mode has nearly no effect on both throughput and average latency due to a limited search scope. However, *TiDedup* with full mode badly affects performance, due to two reasons: (1) cold or deduped objects are accessed frequently in full mode, and, (2) YCSB's request can be blocked during *tier-flush*.

5.3 The impact of average chunk size

Figure 13 shows how much storage space can be saved depending on the average chunk size [46]. We generate 50% of redundant contents by `fio` with varying chunk sizes, then launch the crawler to perform deduplication. In the case of a large chunk (>16K), the amount of reduced data increases rapidly because the number of generated chunks is less than the small chunk's size, so the deduplication job can be done early. Also, we observe that the deduplication ratio does not reach 50% due to additional metadata for deduplication, as shown in Figure 13 (B). Interestingly, CDC generates non-aligned data, unlike FSC; for instance, a 8,200 byte chunk—not aligned by block size (4,096 byte)—can be generated by CDC, causing non-aligned data allocation. As a result, 12 KB is allocated even though the requested object size is 8,200 byte. This aggravates metadata consumption.

5.4 Worst-case recovery performance

We run fio with 8KB random read/write workload—the ratio between reads and writes is 8:2. During the mixed workload, the crawler issues bursty traffic—64 threads submit object reads and *tier_flush* concurrently—to the base tier, and triggers *scrub* using 16 threads. On top of that, 4 out of 36 OSDs are suddenly down during the test, resulting in generating recovery I/Os for data rebalance. Figure 14 (A) and (B) show *TiDedup*'s read and write IOPS over time. Overall, we observe performance fluctuation due to the bursty traffic from the crawler. At around 50 seconds, two OSDs are pulled out from the storage cluster and the performance decreases rapidly due to the recovery I/Os. After 50 seconds, two more OSDs are out and the fluctuation gets worse. However, at around 350 seconds, all four OSDs rejoin the cluster and the performance of foreground I/Os are recovered eventually. Compared to `NoDedup`, even in the worst case, *TiDedup* only reduces the performance by 25% on average. It is noteworthy that we can further mitigate foreground performance drop if the crawler employs a flow control technique in the event of congestion.

5.5 Scrub time

To compare *scrub* time between `Fixed` and *TiDedup*, we run a crawler, which spawns sixteen *scrub* threads (each thread issues six *scrub* requests asynchronously), with varying the number of metadata objects as shown in Figure 15. Each *Scrub* thread searches objects in the chunk tier and reads chunk information as described in Section 4. If the chunk object has a reference to the metadata object, the *scrub* thread reads either all metadata objects (`Fixed`) or a corresponding metadata object (*TiDedup*) in the base tier to check if the reference for chunk object is valid (`Scrub-search`). If the reference is invalid, *TiDedup* fixes the mismatch (`Scrub-repair`). In `Fixed`, *scrub* time increases considerably due to a full scan for metadata objects. *TiDedup* also takes considerable time for *scrub*, but the execution time is significantly reduced because *TiDedup* needs only a single read to check if the referenced metadata object is valid. In the case of one million metadata objects, *TiDedup* takes about four hours to complete, while we could not measure the `Fixed` *scrub* time because even after five days, the job had not been done.

5.6 Snapshot creation and deletion

Figure 16 (A) shows snapshot creation time (ten objects) as the number of deduplicated chunks grows. To measure snapshot creation time, we create deduplicated chunks on a metadata object and issue snapshot creation requests to the metadata object. In `Fixed`, the execution time to create a snapshot increases linearly because the more deduplicated chunks the snapshot has, the more operations (*chunk_create_or_get_ref*) OSD needs to handle. On the other hand, *TiDedup* takes a constant time even thanks to the OID shared reference count.

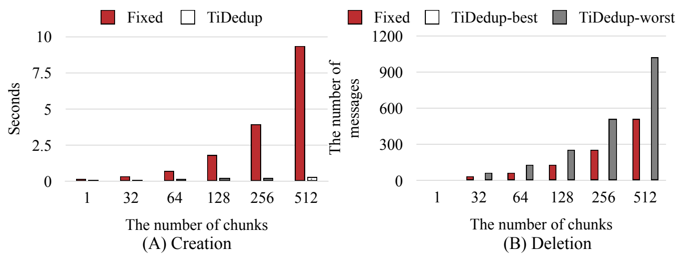


Figure 16: Snapshot creation/deletion comparison.

YAML	Description
dedup-io-mixed	read, write, set_chunk, tier_promote, tier_evict, tier_flush
dedup-io-snap	read, write, set_chunk, tier_promote, tier_evict, tier_flush, snap_create, snap_remove, rollback

Table 3: Integration test description.

Figure 16 (B) shows the number of generated messages from an OSD during snapshot deletion. *TiDedup*-best is the best-case scenario where no chunk differs from adjacent snapshots (e.g., chunk AAA at offset 0 in Figure 8). On the other hand, *TiDedup*-worst represents the extreme case where all chunks are different from adjacent snapshots and both the prior and next snapshots are identical (e.g., chunk DDD at offset 8 in Figure 8). In *TiDedup*-best, no messages are generated. However, compared to `Fixed`, *TiDedup*-worst generates a higher number of messages because *TiDedup* needs to reduce the references due to the same adjacent snapshots. But, these additional messages are proactively generated. Unlike *TiDedup*, `Fixed` would eventually generate the messages if further deletion occurs on either the prior or the next snapshot.

5.7 Integration test

We improve existing stress test coverage in Ceph to make *TiDedup* stable. With yaml as shown in Table 3 and the test code we added, Ceph's integration test framework, called *teuthology* [38], can perform the tests with/-without other tests (e.g., network disconnection and OSD fail) for reliability. Moreover, to ensure reference reliability, the workload generator also checks if the live chunk object's source reference is valid after all operations are done. *TiDedup* passes the combination tests conducted by *teuthology*.

6 Discussions

Small chunk vs. large chunk. Small chunk sizes (< 8KB) would result in numerous small chunk objects and potentially large space overheads. On the other hand, the use of a large chunk size may lead to a lower deduplication ratio. In order to hit a desirable trade-off point, *TiDedup* provides an *estimate* CLI, which shows how much storage space can be saved depending on options, to users. *TiDedup* allows the users to select suitable options (e.g., FSC, CDC, and chunk size) or even decide not to use deduplication at all.

How scalable is crawler? We did not fully address how our crawler design works at scale. In Section 5, we deploy a single application that has multiple threads for crawling. However, if the cluster has a large number of objects, this application may not be able to cover all objects properly. To solve this problem, the system administrator can deploy many crawlers on demand where each crawler is in charge of a sub-dataset of the storage cluster depending on the workload. This task is easily done by using container-based deployment because the crawler is stateless.

Additional space overhead for OID reference. Since the OID-shared reference scheme uses OID to represent a reference, chunk objects should maintain all metadata object's OIDs referring to the chunk objects, requiring more storage space than the reference count method. We limit the number of chunk object's references to a threshold value. *TiDedup* forces OSD to stop performing deduplication on the chunk objects in case their number of references is over the threshold value.

Applicability of *TiDedup*. *TiDedup* relies on two schemes that are prerequisites for integration with other distributed storage systems: (1) two separate address spaces (base and chunk tier), and (2) chunk object lookup by using a hash-based mapping between those two tiers. While the two address spaces scheme is easily applicable to other storage systems, the lookup method is tightly coupled with the hash-based object placement scheme [15, 19, 36, 43].

7 Lessons Learned

Rethinking a tiering mechanism in a distributed storage system with strong consistency. Ceph originally implemented the tiering structure where a background thread reads and migrates objects between tiers for cache tiering, so we anticipated that adding a deduplication feature on top of the existing architecture would be straightforward. However, we needed to consider recovery scenarios against a variety of failure types. Also, the deduplication jobs not only required holding a PG lock for an extended period, especially as the number of objects grew, but also introduced a new lock domain. Note that a strong consistency storage system, such as Ceph, handles I/O operations in strict order while holding locks. Unfortunately, this approach eventually led to an imbalance in I/O loads across different PGs, resulting in significant degradation of user-perceived performance and tail latency. Considering these challenges, we made the decision to deprecate the existing tiering mechanism. Instead, we opted to delegate the responsibility of the tiering mechanism within OSD to other components, such as the crawler.

Deduplication is promising only when data is dedup-able. In a distributed storage system, storing data entails duplicate copies to ensure availability through replication or erasure coding. However, this increases storage space overheads more than we expected due to the following two reasons: (1) additional metadata space for deduplication, and (2) unaligned existing metadata caused by the small size of the new meta-

Table 4: Comparison of previous deduplication architectures and this work.

	TiDedup	**CephDedup** [29]	**Data Domain** [9]	**DupHunter** [49]	**Nitro** [21]	**idedup** [35]
Processing	post	post	post	post	in-line	in-line
Selective dedup	O	X	X	O	X	O
Chunking	CDC, FSC	FSC	CDC	unknown	FSC	FSC
Scale	global	global	local	global	local	local
Interface	file/object/block		file	file	block	block
Storage type	general	general	backup	docker image	general	primary
Implementation	Ceph mainline	research only	proprietary	research only	research only	proprietary

data added for deduplication. Moreover, a general purpose storage system cannot predict in advance whether incoming data is always dedup-able or not. As a result, based on the observation, we decided to perform deduplication only if the storage system is able to secure enough free space after deduplication is done.

Flexible namespace architecture for *TiDedup*. A single global namespace (tier) is not optimal for efficiently handling different types of data streams, as each stream may have its own unique access pattern. Considering that a general-purpose storage system must cater to diverse workloads, we have designed a flexible namespace architecture that allows users to create custom namespaces tailored to their specific workloads. With this architecture, users are able to create one or more custom namespaces, enabling them to handle multiple data streams while maintaining isolation between them. For example, users can align their custom namespaces (base tiers) with the corresponding services, such as object, file, and block, while utilizing a shared chunk tier.

8 Related Work

Although there is rich literature on deduplication [1,2,5,8,13, 27,40,41,48,50,51], few studies evaluate their architecture in terms of scalability in real distributed storage systems and/or open their code for third party reproduction of results. In the next, we touch on the most relevant studies to our work. Table 4 gives a quick summary of their key properties.

Data Domain Cloud Tier [9] proposes a deduplication architecture based on two tiers, where it performs deduplication in the local tier first, and then backs up data in the remote tier. Unlike our work, Data Domain Cloud Tier targets a local storage and does not selectively performs deduplication according to redundancy. *TiDedup* is a cluster-level solution with scalability as key design consideration. To that end, *TiDedup* reduces the fingerprint index lookup overheads by using double hashing [29] and pursues selective deduplication.

DupHunter [49] builds on a multi-layer tier and employs a cache algorithm utilizing domain-specific knowledge (container image registry). Also, it takes a selective deduplication approach. We note that DupHunter targets a specific environment where container images are stored and distributed by docker registry. On the other hand, *TiDedup* is designed for more general environments where file, object, and block services are needed. It is unclear how DupHunter addresses the fingerprint index problem [12,45] and whether the design can coexist with other existing features in a real storage system.

Deduplication could hurt read performance when an object is scattered throughout the cluster. Several studies suggest techniques (e.g., prefetching and rewriting) [3, 11] to overcome degraded read performance. Among them, we believe that caching is the most efficient way to resolve the read performance problem. *TiDedup* exploits a caching technique and migrates chunk objects from chunk tier to base tier if the chunk object is frequently accessed. As a result, hot data will be served quickly with no overhead whereas serving cold data entails forwarding overheads between base and chunk tier.

Inline deduplication [13,21,35] eliminates data redundancy immediately. However, *TiDedup* adopts a post-processing technique along with caching. By doing so, *TiDedup* employs on-demand deduplication, allowing for minimizing performance degradation. We believe that it is more suitable for a general purpose distributed storage system.

9 Conclusion

This paper presents *TiDedup* towards efficient cluster-level deduplication for a general-purpose distributed storage system. *TiDedup* incorporates three new design schemes to overcome the scalability issues found in a prior proposal. We have a complete, fully validated implementation of the design and have integrated *TiDedup* into the Ceph main branch. Our comprehensive evaluation study reveals that *TiDedup* achieves storage space savings without hurting the scalability of Ceph. We hope that our work will become a foundation on which further research and development are undertaken.

Acknowledgement

We would like to thank the anonymous reviewers and our shepherd, Vasily Tarasov, for their valuable comments. This work was supported in part by the National Research Foundation of Korea(NRF) grant funded by the Korea government (MSIT) (No. NRF-2021R1A2C2003618).

References

[1] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–9, 2009.

[2] Z. Cao, H. Wen, X. Ge, J. Ma, J. Diehl, and D. H. C. Du. TDDFS: A tier-aware data deduplication-based file system. *ACM Trans. Storage*, 15(1):4:1–4:26, 2019.

[3] Z. Cao, H. Wen, F. Wu, and D. H. C. Du. ALACC: accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *USENIX Conference on File and Storage Technologies, (FAST)*, pages 309–324, 2018.

[4] CERN. CERN cloud home page. https://clouddocs.web.cern.ch/. Accessed 2023-06-09.

[5] F. Chen, T. Luo, and X. Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 77–90, 2011.

[6] Z. Chen and K. Shen. Ordermergededup: Efficient, failure-consistent deduplication on flash. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 291–299, 2016.

[7] CNCF. CNCF user survey 2020. https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf. Accessed 2023-06-09.

[8] W. Dong, F. Douglis, K. Li, R. H. Patterson, S. Reddy, and P. Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *9th USENIX Conference on File and Storage Technologies (FAST)*, pages 15–29, 2011.

[9] A. Duggal, F. Jenkins, P. Shilane, R. Chinthekindi, R. Shah, and M. Kamat. Data domain cloud tier: Backup here, backup there, deduplicated everywhere! In *USENIX Annual Technical Conference (ATC)*, pages 647–660, 2019.

[10] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta. Primary data deduplication - large scale study and system design. In *USENIX Annual Technical Conference (ATC)*, pages 285–296, 2012.

[11] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, F. Huang, and Q. Liu. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *USENIX Annual Technical Conference (ATC)*, pages 181–192, 2014.

[12] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, Y. Zhang, and Y. Tan. Design tradeoffs for data deduplication performance in backup workloads. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 331–344, 2015.

[13] Y. Fu, H. Jiang, and N. Xiao. A scalable inline cluster deduplication framework for big data protection. In *Middleware 2012*, pages 354–373, 2012.

[14] gke. google kubernetes engine. https://cloud.google.com/kubernetes-engine. Accessed 2023-06-09.

[15] glusterfs. glusterfs home page. https://www.gluster.org/. Accessed 2023-06-09.

[16] K. Jin and E. L. Miller. The effectiveness of deduplication on virtual machine disk images. In *The Israeli Experimental Systems Conference (SYSTOR)*, page 7, 2009.

[17] S. Kaisler, F. Armour, J. A. Espinosa, and W. Money. Big data: Issues and challenges moving forward. In *Hawaii International Conference on System Sciences*, pages 995–1004, 2013.

[18] kubernetes. kubernetes home page. https://kubernetes.io/. Accessed 2023-06-09.

[19] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[20] E. Lee, Y. Han, S. Yang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. How to teach an old file system dog new object store tricks. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.

[21] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace. Nitro: A capacity-optimized SSD cache for primary storage. In *USENIX Annual Technical Conference (ATC)*, pages 501–512, 2014.

[22] H. Liu, W. Ding, Y. Chen, W. Guo, S. Liu, T. Li, M. Zhang, J. Zhao, H. Zhu, and Z. Zhu. CFS: A distributed file system for large scale container platforms. In *International Conference on Management of Data*, page 1729–1742, 2019.

[23] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel. A study on data deduplication in hpc storage systems. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, 2012.

[24] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. In *USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[25] J. Min, D. Yoon, and Y. Won. Efficient deduplication techniques for modern backup operation. *IEEE Transactions on Computers*, 60(6):824–840, 2011.

[26] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 174—187, 2001.

[27] A. Nachman, G. Yadgar, and S. Sheinvald. GoSeed: Generating an optimal seeding plan for deduplicated storage. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 193–207, 2020.

[28] F. Ni and S. Jiang. Rapidcdc: Leveraging duplicate locality to accelerate chunking in cdc-based deduplication systems. In *ACM Symposium on Cloud Computing (SoCC)*, pages 220–232, 2019.

[29] M. Oh, S. Park, J. Yoon, S. Kim, K.-w. Lee, S. Weil, H. Y. Yeom, and M. Jung. Design of global data deduplication for a scale-out distributed storage system. In *IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1063–1073, 2018.

[30] openstack. openstack home page. https://www.openstack.org/. Accessed 2023-06-09.

[31] openstack. openstack user survey. https://www.openstack.org/analytics/. Accessed 2023-06-09.

[32] C. Policroniades and I. Pratt. Alternatives for detecting redundancy in storage systems data. In *USENIX Annual Technical Conference (ATC)*, pages 73–86, 2004.

[33] S. Quinlan and S. Dorward. Venti: A new approach to archival data storage. In *Conference on File and Storage Technologies (FAST)*, pages 89–101, 2002.

[34] D. R.-J. G.-J. Rydning, J. Reinsel, and J. Gantz. The digitization of the world from edge to core. *Framingham: International Data Corporation*, 16, 2018.

[35] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti. idedup: latency-aware, inline data deduplication for primary storage. In *USENIX conference on File and Storage Technologies (FAST)*, page 24, 2012.

[36] Swift. Welcome to swift's documentation! https://docs.openstack.org/swift/latest/. Accessed 2023-06-09.

[37] TencentCloud. Tencent cloud home page. https://www.tencentcloud.com/solutions/tstack. Accessed 2023-06-09.

[38] Teuthology. Teuthology GitHub. https://github.com/ceph/teuthology. Accessed 2023-06-09.

[39] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of backup workloads in production systems. In *USENIX conference on File and Storage Technologies (FAST)*, page 4, 2012.

[40] C. Wang, Q. Wei, J. Yang, C. Chen, Y. Yang, and M. Xue. NV-Dedup: high-performance inline deduplication for non-volatile memory. *IEEE Transactions on Computers*, 67(5):658–671, 2017.

[41] Q. Wang, J. Li, W. Xia, E. Kruus, B. Debnath, and P. P. Lee. Austere flash caching with deduplication and compression. In *USENIX Annual Technical Conference (ATC)*, pages 713–726, 2020.

[42] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–320, 2006.

[43] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *ACM/IEEE Conference on Supercomputing*, pages 31–31, 2006.

[44] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.

[45] W. Xia, H. Jiang, D. Feng, and Y. Hua. Silo: A similarity-locality based Near-Exact deduplication scheme with low RAM overhead and high throughput. In *USENIX Annual Technical Conference (ATC)*, 2011.

[46] W. Xia, Y. Zhou, H. Jiang, D. Feng, Y. Hua, Y. Hu, Q. Liu, and Y. Zhang. FastCDC: A fast and efficient content-defined chunking approach for data deduplication. In *USENIX Annual Technical Conference (ATC)*, pages 101–114, 2016.

[47] Y. Zhang, H. Jiang, D. Feng, W. Xia, M. Fu, F. Huang, and Y. Zhou. AE: an asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication. In *IEEE Conference on Computer Communications (INFOCOM)*, pages 1337–1345, 2015.

[48] Y. Zhang, W. Xia, D. Feng, H. Jiang, Y. Hua, and Q. Wang. Finesse: Fine-Grained feature locality based fast resemblance detection for Post-Deduplication delta compression. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 121–128, 2019.

[49] N. Zhao, H. Albahar, S. Abraham, K. Chen, V. Tarasov, D. Skourtis, L. Rupprecht, A. Anwar, and A. R. Butt. DupHunter: Flexible high-performance deduplication for docker registries. In *USENIX Annual Technical Conference (ATC)*, pages 769–783, 2020.

[50] X. Zou, W. Xia, P. Shilane, H. Zhang, and X. Wang. Building a high-performance fine-grained deduplication framework for backup storage with high deduplication ratio. In *USENIX Annual Technical Conference (ATC)*, pages 19–36, 2022.

[51] X. Zou, J. Yuan, P. Shilane, W. Xia, H. Zhang, and X. Wang. The dilemma between deduplication and locality: Can both be achieved? In *USENIX Conference on File and Storage Technologies (FAST)*, pages 171–185, 2021.

A Artifact Appendix

Abstract

We provide the artifact that includes the source code and instructions to explain how to run *TiDedup* on Ceph. The artifact also includes a document to describe how *TiDedup* is applied to Ceph.

Scope

TiDedup is a cluster-level deduplication architecture for Ceph, so the goal of the artifact is to describe how to perform deduplication on Ceph cluster using either the artifact or mainline Ceph. As explained in README.md, we provide instructions to build, deploy, and run *TiDedup* on Ceph. In addition, the README.md also provides an explanation of how to run *TiDedup* using the latest Ceph without the artifact.

Contents

The artifact contains *TiDedup*'s source code integrated into Ceph and a README.md file to build source code and run Ceph while enabling deduplication.

Hosting

TiDedup is available at https://github.com/ssdohammer-sl/ceph/tree/tidedup with detailed instructions. Since *TiDedup* has been merged to Ceph mainline, the source code is also available at https://github.com/ceph/ceph.