Cheetah: Metadata Aggregation for Fast Object Storage without Distributed Ordering

Yiming Zhang
[1]Shanghai Key Laboratory of Trusted Data Circulation, Governance and Web3, [2]NICE XLab, XMU
China
sdiris@gmail.com

Li Wang
KylinSoft
Changsha, Hunan, China
laurence.liwang@gmail.com

Shengyun Liu
SJTU
Shanghai, China
shengyun.liu@sjtu.edu.cn

Shun Gai, Haonan Wang
NICE XLab
Xiamen, Fujian, China
{gaishun,wanghaonan}@nicexlab.com

Xin Yao, Meiling Wang
Huawei Theory Lab
Hong Kong, China
{yao.xin1,wangmeiling17}@huawei.com

Kai Chen
HKUST
Hong Kong, China
kaichen@cse.ust.hk

Dongsheng Li
NUDT
Changsha, Hunan, China
dsli@nudt.edu.cn

Jiwu Shu
Tsinghua University
Beijing, China
shujw@tsinghua.edu.cn

Abstract

Object stores usually maintain the mapping of objects to data servers' disk volumes (referred to as *volume metadata*) in a central directory, while storing the object data's in-volume offsets (referred to as *offset metadata*) together with the data on data servers. Unfortunately, the separation between volume/offset metadata complicates the processing of an object `put`: to ensure consistency, the multiple writes of the object's volume/offset metadata and object data have to be orchestrated in a particular *order*, which severely lowers object I/O performance. We propose a write-optimal structure called MetaX that aggregates all metadata of a `put`, including both volume and offset metadata as well as other meta information such as data checksum and temporary meta-log. Based on MetaX, we design the *Cheetah* object store, which organizes object storage into rich metadata storage (on meta servers) and raw data storage (on data servers). Cheetah removes the *distributed ordering* constraint on the multiple metadata/data writes by enforcing *local atomicity* of writing MetaX, while still ensuring consistency. Evaluation shows that Cheetah significantly outperforms existing object stores.

EuroSys '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/25/03
https://doi.org/10.1145/3689031.3696080

CCS Concepts: • **Computer systems organization** → **Cloud computing**; **Reliability**.

Keywords: Metadata aggregation, object storage, ordering

ACM Reference Format:
Yiming Zhang, Li Wang, Shengyun Liu, Shun Gai, Haonan Wang, Xin Yao, Meiling Wang, Kai Chen, Dongsheng Li, and Jiwu Shu. 2025. Cheetah: Metadata Aggregation for Fast Object Storage without Distributed Ordering. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3689031.3696080

1 Introduction

Object stores provide a simple interface of `put`/`get`/`delete` for applications to write/read/remove objects, but disallow overwriting existing objects (*i.e.*, immutability). As the price of solid-state drives (SSDs) has dropped drastically [18, 19], latency-sensitive applications that have small total latency budgets (usually hundreds of milliseconds) store both metadata and data of their *small* objects (ranging from several KBs to a few hundred KBs) on SSDs to allow fast access.

There are two categories of approaches for placing objects onto disk volumes, namely, *directory-based mapping* [21, 37] that is maintained by a central directory service, and *hash-based mapping* [45, 49, 50] that is consistently calculated in a decentralized manner. Commercial object stores with fast-expanding business often prefer directory-based approaches to hash-based ones, as these object stores require flexible management like *migration-free expansion* [47]. For instance, Haystack [21] uses a directory to flexibly place objects onto volumes; and Tectonic [42] supports file-based object storage and explicitly controls the mapping of chunk files to volumes.

Although widely used for read-dominant applications [1, 14], existing directory-based object stores [21, 37, 42] cannot efficiently support write operations for *small* objects. This is because they separately maintain the object-to-volume mapping (*i.e.*, *volume metadata*) on directory servers and the in-volume file offset (*i.e.*, *offset metadata*) on data servers, which complicates the processing of object `put`: the volume metadata and offset metadata are updated separately, and (for crash consistency) the writes of volume/offset metadata and data must be orchestrated in a particular *order*. The distributed ordering constraint severely lowers object write performance, which narrows the applicability of object.

We design a write-optimal structure called MetaX that aggregates all metadata of an object `put`, including volume metadata, offset metadata as well as other meta information like data checksum and temporary meta-log. The goal of metadata *aggregation* is for atomic access and update of the metadata of each object, which enables us to remove the distributed ordering constraint and ensure consistency by enforcing local atomicity of writing the aggregated MetaX structure. In contrast, traditional metadata *disaggregation* of distributed filesystems (like ADLS [43], HopsFS [36], Tectonic [42], and InfiniFS [2]) is mainly for scalability/efficiency of metadata maintenance of the entire filesystem. For instance, Tectonic disaggregates the filesystem metadata into naming, file, and block layers each being maintained in a separate sharded key-value (KV) store.

Based on MetaX, we design an efficient object store called Cheetah, which organizes object storage into *rich metadata storage* on meta servers and *raw data storage* on data servers. Cheetah can write metadata (on meta servers) and object data (on data servers) in parallel without distributed ordering, and access all metadata of an object as a whole (on meta servers) avoiding the overhead of separately accessing offset metadata, which benefits object `put`/`get`/`delete` operations.

Metadata aggregation brings the following challenges.

First, as Cheetah aggregates all metadata into MetaX, metadata storage of an ever-growing number of objects requires either directory-based maintenance that lowers lookup performance, or hash-based maintenance that causes data migration after system expansion. To tackle this challenge, We extend CRUSH [50] to design a hybrid mapping architecture. Each placement group (PG) has exactly one corresponding volume group (VG), and objects in a PG can be first mapped onto the PG's meta server and then to the volumes in the PG's VG, realizing both migration-free expansion of data storage and high scalability of metadata storage.

Second, parallel metadata and data writes of objects complicate reliability/consistency guarantees. We carefully organize the MetaX structure into a set of KVs, and design Cheetah's replication and recovery mechanisms using a combination of topology maps, leases, and view numbers [26, 39]. We prove that Cheetah guarantees reliability and consistency (owing to object immutability) under various crashes.

Cheetah efficiently supports not only read-dominant workloads [9, 14] but also write-dominant and read-write-hybrid workloads [30] that frequently `put`/`delete` objects, thus effectively broadening the applicability of directory-based object storage. We compare Cheetah to the state-of-the-art directory based object stores (represented by Haystack [21]) as well as hash-based object stores (represented by Ceph [38, 49]). Cheetah significantly improves the I/O performance of small objects in both latency and throughput.

2 Background & Motivation

2.1 Object Placement

An object usually has a unique name, actual data, and optionally some attributes like times and locations. Object stores place objects onto appropriate volumes either by adopting directory-based mapping or by using hash-based calculation. Theoretically, hash-based methods (represented by CRUSH [50]) have better performance, as they directly calculate the target volumes without accessing directories.

However, CRUSH-like hash-based placement suffers from data migration and performance degradation when expanding the capacity, which is common for commercial systems with fast-expanding business and ever-increasing storage demand [47]. Although this problem can be temporarily mitigated by limiting the migration rate [49], all objects still have to be *eventually* placed onto the calculated volumes, resulting in an even longer period of performance degradation.

Other hash-based methods (like ring-hash [45]) not only suffer from similar migration with more severe imbalance, but also are unable to model the cluster hierarchy. MapX [47] uses time-dimension mapping (from object creation times to expansion times) for object-based *block* storage (Ceph-RBD [8]) to avoid migration. But it cannot support object storage due to the overwhelming per-object timestamp overhead.

In contrast, directory-based placement can easily realize migration-free expansions by managing the object-to-volume mapping, keeping existing objects unaffected after expansions, and placing new objects onto new volumes. Further, directory-based placement also allows flexible management [21, 37] such as auditing and remapping.

2.2 Problem of Directory-Based Mapping

In directory-based object stores, a directory service maintains the mapping from objects to volumes (*i.e.*, *volume metadata*, *a.k.a.* application metadata [21]). To reduce the per-object metadata overhead suffered by small-file-based object stores (like Swift [16] and S3 [5]) where each object is stored as a small file, modern object stores [21, 37, 42] usually optimize storage organization by appending objects to large volume files, which needs to maintain the local mapping from an object to its file offset (*i.e.*, in-volume *offset metadata*).

The separation of the volume metadata (in directories) and the offset metadata and actual data (on data servers)

greatly complicates the processing of object write operations. First, the volume metadata and offset metadata are written separately. Second, the writes of volume/offset metadata and/or data have to be orchestrated in a particular *order* [24] (discussed in detail in the next subsection). This is to ensure (i) the distributed volume and offset metadata to be consistent (*i.e.*, metadata consistency), and (ii) the metadata to be consistent with its data (*i.e.*, data consistency).

Separate metadata writes severely affects the performance of object `put` and `delete`. For instance, object `delete` operations in Haystack [21] and Tectonic [42] include several steps: (i) accessing the directory to query the volume metadata, (ii) accessing data servers to update relevant offset metadata, and (iii) accessing the directory again to update the volume metadata. Further, Haystack optimizes the processing of object `put` by having its data servers synchronously update the offset metadata in an in-memory KV and asynchronously checkpoint the in-memory KV to an on-disk index file. Although being effective for read-intensive workloads, this optimization is not suitable for write-intensive workloads, where asynchronous checkpointing will cause the persistent index file severely lagged behind the in-memory KV and make failure recovery difficult or even impossible.

2.3 Distributed Write Ordering

In the event of a crash, it is critical to reason about which piece of data/metadata was already persisted and which was not, for taking corrective recovery measures accordingly. Crash consistency requires distributed ordering for writing the volume metadata in directories and the offset metadata and data on data servers. Such distributed ordering introduces waits into the critical path of `put`/`delete`, and thus lowers I/O performance especially for *small* objects that have relatively-low overhead in writing objects' actual data.

We briefly discuss the inefficiency of distributed write ordering by taking Haystack [21] as a representative, which maintains the volume metadata in a directory service and the offset metadata (together with data) on data servers. Other directory-based object stores like Tectonic [42] have similar designs of *separate* metadata storage. Haystack includes (i) a directory service served by meta servers, (ii) a data server cluster, and (iii) individual clients each using a proxy as a portal to interact with meta/data servers. For clarity, we assume *one-way replication* where each object has only one replica so that it can prevent data loss from temporary crashes but not from permanent disk failures or machine crashes. Object attributes (such as times/locations) are viewed as object data.

Objects are immutable, *i.e.*, an object is written only once and never overwritten [21]. Once an object has been deleted, a new immutable object with the same name may be created again. This simplifies the ordering constraint between *multiple* `put` requests, because a later `put` will not overwrite an existing object, allowing our discussion to focus on the ordering of the multiple writes of an *individual* `put`.

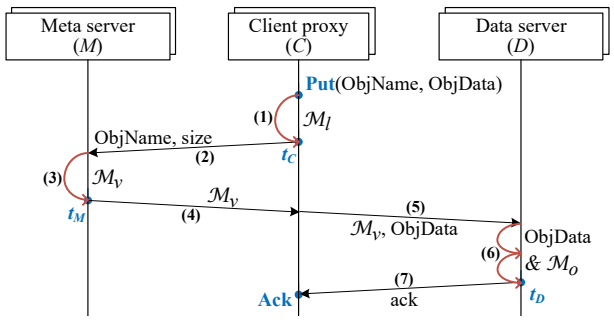


Figure 1. Distributed write ordering for crash consistency: write-ahead meta-log ($\mathcal{M}_l$) on $C \rightarrow$ volume metadata ($\mathcal{M}_v$) on $M \rightarrow$ object data & offset metadata ($\mathcal{M}_o$) on D.

Considering possible crashes of client proxies (issuing the `put`), meta servers (storing volume metadata) and data servers (storing offset metadata and data), the basic processing of `put` in Haystack is shown in Fig. 1, where we enforce distributed ordering for metadata and data consistency [24].

(1) After receiving a `put`, the client proxy C persists the write-ahead meta-log ($\mathcal{M}_l$) including the object *name*, selected directory meta server (M), and data checksum (c).

(2) C sends the object *name* and *size* to M.

(3) The directory meta server M chooses a volume and map the object to it, and persists the mapping information (*i.e.*, volume metadata denoted as $\mathcal{M}_v$).

(4) M returns $\mathcal{M}_v$ to C after persistence. Note that M cannot return $\mathcal{M}_v$ before it is persisted. Otherwise, data server (D) might receive $\mathcal{M}_v$ and data too early: if the data is successfully persisted on D and both C and M fail before $\mathcal{M}_v$ is persisted, then the persisted data will become orphan.

(5) C sends object data and $\mathcal{M}_v$ to the data server D.

(6) The data server D allocates space for received object data at the end of the volume file (specified by $\mathcal{M}_v$), and persists the data and the allocation (*i.e.*, offset metadata $\mathcal{M}_o$).

(7) D returns to proxy C. At this point, the meta-log of the client proxy C can be cleaned, and the `put` is *committed*.

The sequence of disk writes (each being denoted as a "ς" in Fig. 1) takes place in the following order: $\mathcal{M}_l$ on $C \rightarrow \mathcal{M}_v$ on $M \rightarrow$ object data & $\mathcal{M}_o$ on D, where $\rightarrow$ represents the *distributed ordering* relationship. Note that the ordering of local writes (of data and $\mathcal{M}_o$) on D has been well studied [23, 24] thus being omitted here.

Consider a power failure that possibly takes place at any time t disabling any servers. The first $\rightarrow$ enables C to track uncommitted `put` requests: if a failure happens at $t > t_C$, then C can check whether a `put` in $\mathcal{M}_l$ is finished by first querying M with *name* and then querying D with $\mathcal{M}_v$. The second $\rightarrow$ ensures that persisted data on D will not become orphan due to a failure at time $t > t_D$ because $t_D > t_M$. $\mathcal{M}_v$ can be safely written on M without worrying that a failure at $t_M < t < t_D$ can both fail the current `put` and invalidate the metadata of an existing object with the same

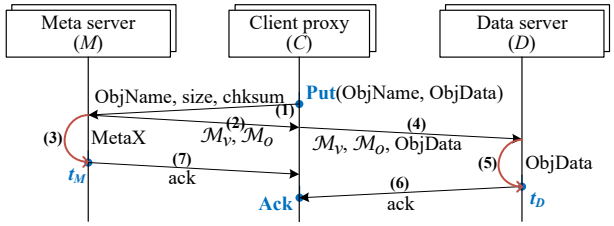


Figure 2. Parallel writes (without ordering) of aggregated MetaX (= $\mathcal{M}_l + \mathcal{M}_v + \mathcal{M}_o$) on M and object data on D.

name, as Facebook's upper-layer applications (*e.g.*, photo uploader) ensure object immutability. Note that an optional optimization [20] is that the meta server M sends and persists volume metadata $\mathcal{M}_v$ in parallel, and the logging on C of step (1) can be postponed until receiving $\mathcal{M}_v$. Then, C logs $\mathcal{M}_v$, M, and the checksum (c), which allows for removing the ordering of writes on M and D. However, this optimization still has ordering constraint for the writes on C and M/D, thus suffering similar problem as the original Haystack.

3 Metadata Aggregation with MetaX

3.1 Rich Meta Service

We extend the traditional *thin directory service* (on directory meta server M in Fig. 1) to the *rich meta service* (on meta server M in Fig. 2). Accordingly, the client proxy performs as a portal for forwarding requests and returning results; and the data servers provide *raw* data service that is agnostic to the objects and simply reads/writes specified data blocks. Note that this agnosticism will not affect the internal management of SSD's flash translation layer (FTL), *e.g.*, for avoiding wear and reducing fragmentation.

The meta service is responsible for maintaining (i) the mapping of objects to volumes, (*volume metadata* $\mathcal{M}_v$), (ii) the mapping of objects to in-volume blocks (*offset metadata* $\mathcal{M}_o$) with data checksum (c), and (iii) the meta-log ($\mathcal{M}_l$) including object *name*, client proxy (C), etc. We refer to all meta information of a `put` ($\mathcal{M}_v$, $\mathcal{M}_o$, $\mathcal{M}_l$) as MetaX.

3.2 Consistency without Distributed Ordering

As MetaX integrates all pieces of metadata/meta-log of a `put` and eliminates possibilities of inconsistency between them, the writes of MetaX on meta servers and object data on data servers can be processed in parallel without ordering, as depicted in Fig. 2 (where we assume one-way replication).

(1) The client proxy C sends the object *name*, *size* and checksum (c) to a selected meta server M.

(2) Meta server M allocates the volume ($\mathcal{M}_v$) and in-volume available blocks ($\mathcal{M}_o$), and sends $\mathcal{M}_v$ and $\mathcal{M}_o$ to C.

(3) Meanwhile, M also persists MetaX (= $\mathcal{M}_v + \mathcal{M}_o + \mathcal{M}_l$) in parallel. Note that the meta-log ($\mathcal{M}_l$) is part of MetaX.

(4) C sends the object data with the allocation result ($\mathcal{M}_v$ and $\mathcal{M}_o$) to the corresponding data server D.

(5) D persists the raw data according to $\mathcal{M}_v$ and $\mathcal{M}_o$.

(6) After persistence, D returns ack (with checksum) to C.

(7) M also returns ack to C after persistence of MetaX.

Note that in Step (3) of Fig. 1, the client persists a write-ahead meta-log to enable the client to handle uncommitted `put`. In contrast, in Fig. 2 we can rely on the meta server to handle it, as we have aggregated all metadata into MetaX. Here we take one-way replication and only consider temporary crashes from which the meta server can always recover.

Only if both M and D correctly return to C, is the `put` request *committed*. Otherwise, *e.g.*, if M crashes before persisting MetaX, the `put` is uncommitted and the problem will be addressed (§5). The atomicity of MetaX simplifies consistency and removes the ordering requirement for the two separate writes (MetaX on M and object data on D). See §5.3 for a thorough analysis of consistency guarantees, where a manager maintains the topology map and the failures of meta/data/proxy servers can be (consistently) known.

The parallel processing in Fig. 2 leverages immutability to avoid overwrite-caused inconsistency. Note that parallel processing might cause inconsistency without the promise of immutability. For instance, if both client proxy C and data server D crash at time $t_M < t < t_D$, then an existing object with the same *name* as the ongoing-`put` object will lose its metadata, because MetaX has been persisted on M but object data is not yet on D. In practice, the maintenance of MetaX is more complex than in the simplified scenario (Fig. 2), in that Cheetah must not only achieve high I/O performance, but also provide high reliability/scalability/availability (§4).

4 Cheetah Object Store

4.1 Architecture

Cheetah consists of a manager cluster, a data server cluster, and a meta server cluster. Each client has a client proxy.

Manager cluster contains an odd number of manager server processes jointly running Raft [41] as one reliable central *system manager*, which maintains a consistent topology map and periodically updates the topology as well as other global information to other servers, as detailed in §5.1.

Data server cluster consists of data storage machines installing disks divided into physical volumes each with one data server process to provide raw data storage service.

Meta server cluster consists of meta storage machines installing disks and running meta server processes to maintain the MetaX structure and provide rich meta service.

Client proxy (as a portal) provides an interface of `put`, `get`, and `delete`. Client proxies are not replicated.

Cheetah adopts n-way replication (storing n replicas) for durability of both data and metadata, but the replicas for data and for metadata are organized in different ways. (i) For data storage, Cheetah first divides a disk into *physical* volumes (*e.g.*, a one-TB disk can be divided into ten 100-GB physical volumes), and then groups physical volumes into *logical* volumes [21]. A logical volume (Fig. 3) consists of n

physical volumes which store exactly the same raw data. This allows a disk's physical volumes to be replicated to many other disks for parallel failure recovery. (ii) For metadata, Cheetah adopts CRUSH for organization and replication.

4.2 Hybrid Mapping

The MetaX structure aggregates all metadata, which increases the overhead of the meta service thus introducing scalability challenges. Not only the data server cluster but also the meta server cluster needs expansion to satisfy the rapidly-growing storage demand. It would be inefficient to rely on an extra "metadata directory", which introduces another level of indirection, to maintain the mapping from objects to meta servers. Therefore, we propose to adopt the hash-based CRUSH algorithm [50] to directly calculate the n target meta servers for each given object. CRUSH models the cluster topology as a logical tree and introduces *placement groups* (PGs) to facilitate the object-to-server mapping: an object is first mapped to a PG by calculating PGID = *Hash*(*name*) mod (*number* of PGs); and then a PG is mapped to its corresponding servers by following specific selection rules top-down in the tree. As shown in Fig. 3, we adopt CRUSH to (i) organize objects into placement groups (PGs) and (ii) map each PG to one primary and $n-1$ backup meta servers. Primary meta servers are responsible to allocate not only logical volumes but also in-volume space for object data. It is common for two PGs with the same primary meta server to have different backup meta servers, and vice versa.

Applying CRUSH to calculate meta servers inevitably causes metadata migration when expanding the meta server cluster, which is acceptable for Cheetah as the size of metadata is orders of magnitude smaller than that of data. What is important is to prevent the expansion of *meta* server cluster from causing migration of object *data*. Counterintuitively, it is nontrivial to achieve this. When adding/removing meta servers, PGs (together with objects' metadata) will be moved to the re-calculated meta servers. This requires each PG to have its *own* group of logical volumes *exclusively* managed by the PG's primary meta server. Otherwise, suppose that two PGs have a common logical volume (vol_0) managed by a primary meta server. After expansion, the two PGs might be "CRUSHed" to two different primary meta servers both of which manage vol_0, making it difficult to ensure consistency.

To address this problem, we organize the logical volumes into *volume groups* (VGs), each corresponding to one PG. The *system manager* maintains the membership of logical volumes in VGs and periodically updates the information (as part of the system topology map) to all servers. Fig. 3 illustrates the mapping procedure from objects to in-volume blocks. Objects are organized into PGs which are "CRUSHed" to meta servers via calculation. When an object of a PG is `put` to Cheetah, the calculated primary meta server first selects a logical volume from the PG's VG, and then allocates available blocks in the logical volume for the object data.

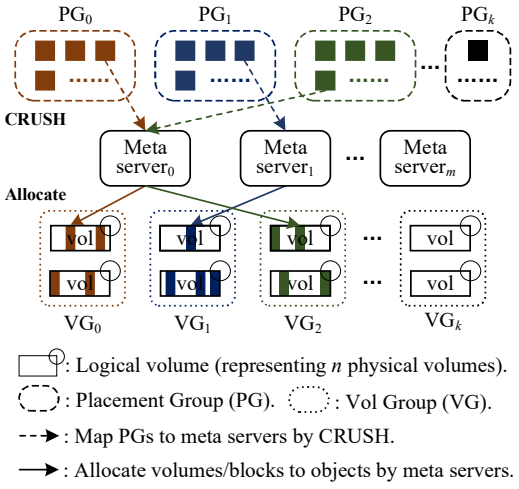


Figure 3. A PG is "CRUSHed" onto a (primary) meta server M, which assigns each object in the PG with a logical volume (from the PG's VG managed by M) and in-volume blocks.

The introduction of VGs avoids object data migration after expansions. If the data server cluster expands, the manager servers will add the new logical volumes to the existing VGs allowing new objects to be assigned to new volumes; if the meta server cluster expands, some PGs might be remapped to the new meta servers (by CRUSH) while the PGs' corresponding VGs will also be managed by the new meta servers without data migration, as neither the mapping from objects to PGs nor the membership of volumes in VGs has changed.

4.3 Object Storage

4.3.1 Parallel metadata & Data Writes for Object Put.

A `put` request includes the object name $\mathcal{N}$ and data $\mathcal{D}$. As shown in Pseudocode 1 and Fig. 4 where both meta and data servers take n-way replication, Cheetah processes `put` by extending the basic procedure of parallel writes of Fig. 2. Failures will be handled through a recovery process (§5.3).

(1) Line 2. The client proxy (C) calculates the primary meta server (M_1) of the object's PG via CRUSH. C also computes the data checksum and generates a unique request ID (*reqid*). Then C sends the object name $\mathcal{N}$, size, checksum (c), and *reqid* to the primary meta server (M_1).

(2) Lines 3~4. The primary meta server (M_1) takes the following two substeps after receiving the request.

(i) M_1 selects a logical volume (ID = *lvid*) from the PG's VG (*i.e.*, volume metadata $\mathcal{M}_v$ = *lvid*). It then allocates raw blocks on the selected volume for storing object data using a bitmap-based allocator [6]. The result is recorded as a list of extents (*i.e.*, offset metadata $\mathcal{M}_o$ = *extents*) of the blocks.

(ii) The MetaX structure (§3.1) consists of volume metadata $\mathcal{M}_v$, offset metadata $\mathcal{M}_o$, and meta-log ($\mathcal{M}_l$) that records the name, client proxy, and checksum. Furthermore, since metadata is replicated at the granularity of PGs (§4.2), we also

Pseudocode 1 Parallel metadata and data writes for Put

```
1:  procedure PUT(ObjName N, Data D, ClientProxy C)
    ▷ Meta servers M1, M2, M3 = CRUSH(PGID of N)
2:      C sends N, object size, checksum, reqid to M1
3:      M1 allocates volume (Mv) and in-volume space (Mo)
        for object N and returns Mv and Mo to C
    ▷ Lines 4~6 and Lines 7~9 are performed in parallel
4:      M1 sends METAX to M2, M3 and persists METAX
5:      M2 and M3 persist METAX and return to M1
6:      M1 returns to C and makes object N pending
7:      C sends (D, Mv, Mo) to data servers D1, D2, D3
8:      D1, D2, D3 write D to disks specified by Mv, Mo
9:      D1, D2, D3 return to C and C acks (i.e., committed)
10:     C notifies M1 that object N committed
11: end procedure
```

log the PG in $\mathcal{M}_l$ for meta server crash recovery (discussed in §5.2). M_1 performs the following operations in parallel:

(a) returning $\mathcal{M}_v$, $\mathcal{M}_o$ to C;

(b) sending MetaX to backup meta servers (M_2, M_3);

(c) locally persisting MetaX.

(3) Lines 5~6. M_2 and M_3 persist the received MetaX and return acks to M_1. After persisting MetaX and receiving all acks, M_1 returns ack to C to confirm successes of persistence on all meta servers. Then, it makes object $\mathcal{N}$ pending (*i.e.*, invisible until the put is committed), for avoiding inconsistency of two get requests (discussed later in Step (6)).

(4) Line 7. C looks for the logical volume (with ID = *lvid*) in its local topology map, and sends the write request (object data $\mathcal{D}$, $\mathcal{M}_v$, and $\mathcal{M}_o$) to the responsible data servers ($D_1 \sim D_3$) of the three physical volumes of the logical volume.

(5) Lines 8~9. $D_1 \sim D_3$ persist object data to the blocks ($\mathcal{M}_o$ = *extents*) on the physical volumes of the logical volume ($\mathcal{M}_v$ = *lvid*), and return acks to C. After receiving acks from both M_1 and $D_1 \sim D_3$, C acknowledges Success. At this point the put request is said to be *committed*.

(6) Line 10. C notifies M_1 the commitment and M_1 makes object $\mathcal{N}$ visible to serve get requests for $\mathcal{N}$.

M_1 cannot serve get requests for $\mathcal{N}$ before M_1 is notified. For instance, consider a put followed by two get. For the put, assuming Cheetah has succeeded in updating meta servers but has not yet completed data writes to all data servers. In this case, the put is not committed and M_1 has to make $\mathcal{N}$ pending. Otherwise, it is possible that the first get reads the finished data server and succeeds, while the second reads the unfinished server and fails causing inconsistency. A fire-and-forget notification is enough, as the meta server M will periodically check uncommitted meta-logs. If M encounters a pending get, it will wait. If M fails with pending meta-logs, after recovery each pending meta-log will be checked. There is no pending in Fig. 2, as it assumes one-way replication.

There is no distributed ordering for the writes in Steps (2)ii(c), (3), and (5) in Fig. 4, which are performed in parallel

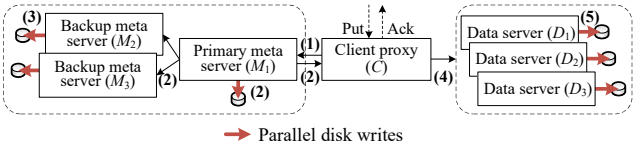


Figure 4. Object put of Cheetah (taking 3-way replication for both meta and data servers). Meta data writes **(2)(3)** and data writes **(4)(5)** execute *in parallel*.

on the n meta servers and n data servers. §5 will discuss how Cheetah ensures consistency with local write atomicity of MetaX on meta servers. For efficiency, M_1 does not immediately clean the meta-log ($\mathcal{M}_l$) of the put. Instead, it periodically cleans the committed meta-logs, checks the uncommitted meta-logs and revokes failed writes (§5.3).

Also note that although immutability disallows overwrites, an object can be *updated* by deleting it and then putting a new one with the same name [21]. An update is equal to an overwrite only if the delete and put can be done atomically.

4.3.2 Object Get.

A get request is processed as follows.

(1) The client proxy (C) calculates the primary meta server (M_1) for object name $\mathcal{N}$ via CRUSH, and queries M_1.

(2) M_1 searches locally for $\mathcal{N}$ and retrieves the metadata ($\mathcal{M}_v$ = *lvid*, $\mathcal{M}_o$ = *extents*, and checksum *c*). If $\mathcal{N}$ is pending, M_1 will check the n data servers whether the data of object $\mathcal{N}$ has been persisted on them, which has not yet been notified to M_1 because the client proxy C may fail after Line 9 in Pseudocode 1. If so, M_1 will try to contact that suspicious client proxy and start recovery if necessary (see §5.3). Otherwise, M_1 returns *lvid*, *extents* and *c* to C.

(3) C looks up the logical volume (with ID = *lvid*) in its local topology map, and then sends a read request with metadata ($\mathcal{M}_v$ = *lvid*, $\mathcal{M}_o$ = *extents*) to the data server of one of the n physical volumes of the logical volume.

(4) The data server reads the blocks ($\mathcal{M}_o$ = *extents*) from the logical volume ($\mathcal{M}_v$ = *lvid*), and returns the data to C.

(5) C verifies the data and (if correct) returns.

4.3.3 Object Delete.

Traditional object storage systems either append objects to large files (*e.g.*, in Tectonic and Haystack), or allocate each object with a small file (*e.g.*, in Swift [16] and MinIO [15]), neither of which is satisfactory for object put/delete performance because their disk space management causes high metadata overhead.

From the perspective of delete, appending objects to files is a lightweight approach for data servers as a delete is simply setting a flag in the offset metadata, but doing so still requires compaction [21] where the space of deleted objects is reclaimed by moving the remaining objects to new files. Unfortunately, similar to expansion-caused data migration in hash-based object storage [47], compaction essentially causes I/O amplification thus lowing the overall I/O performance even being conducted in the background. Compaction is only appropriate for workloads with daily idle windows.

In contrast, one-file-per-object allocation needs no compaction, but it is costly in processing normal object I/O requests because of nontrivial filesystem overhead.

Owing to MetaX, Cheetah's data servers are object agnostic and provide ultralight data service where raw data blocks can be directly accessed without intermediate file abstraction. This allows Cheetah to (i) process `delete` by simply updating metadata on meta servers and (ii) largely reduce compaction. A `delete` request is processed as follows.

(1) The client proxy (C) gets the primary meta server (M_1) for object *name* via CRUSH, and sends `delete` to M_1.

(2) M_1 looks up the MetaX record ($= \mathcal{M}_v + \mathcal{M}_o + \mathcal{M}_l$) for object name $\mathcal{N}$ in its local storage. M_1 then deletes the record, and updates/flushes the bitmap (used by the space allocator) of the logical volume (specified by $\mathcal{M}_v$ = *lvid*) by clearing the bits (specified by $\mathcal{M}_o$ = *extents*). Meanwhile, `delete` is also sent to the backup meta servers and processed similarly.

(3) After receiving acks from all backup meta servers, M_1 returns to C, which finally acknowledges the `delete`.

Compared to file-append allocation of Tectonic/Haystack, Cheetah's raw-block-based allocation not only saves processing on data servers but also allows immediate reuse of the reclaimed space once the corresponding bits of the deleted object data are updated in the bitmap. Neither Tectonic nor Haystack can simply adopt Cheetah's immediate reclamation, because they manage the disk space using chunk files [21, 42] and thus the filesystem overhead of reclamation can severely affect normal I/O performance.

For new objects, the relatively-small object sizes allow us to find appropriate reclaimed blocks with marginal fragmentation, rather than allocating new blocks. Therefore, Cheetah is particularly suitable for write-intensive workloads that frequently perform (unpredictable) `put`/`delete` operations and have no daily idle windows for compaction. Note that Cheetah can also resort to Haystack-style compaction if the overall fragmentation is high, which is rare in practice (§6.4).

5 Recovery

5.1 Overview

A manager cluster, which consists of an odd number of servers jointly running Raft as a reliable system manager, is responsible for managing a global *topology map*, so that Cheetah can use n-way replication (storing metadata/data on n meta/data servers) to tolerate $n-1$ simultaneous meta server crashes and $n-1$ simultaneous data server crashes while ensuring availability, with partial synchrony assumption due to FLP impossibility [25]. The topology map maintains (i) the information of data/meta servers and (ii) the logical volumes of each PG's VG as well as the logical-to-physical volumes mapping. The manager also maintains (iii) the view numbers [39] and (iv) the leases [27].

Servers and client proxies must share the same topology map. To this end, the system manager maintains a view number which is incremented every time the topology map changes. The view number is disseminated to all servers and client proxies, and will be piggybacked with every request to reach a consensus about the current topology map. A request can be processed only if the server has a matching view number. The lagged-behind server will update its view number and topology map coordinated by the manager.

If the manager cannot collect heartbeats [29] from any meta/data server timely, it starts updating the topology, removing the problematic meta servers, and replacing the problematic data servers with new ones of healthy physical volumes within the same VG. The manager will increase the view number by one and disseminate the new map with the view number. It will ask the existing or new primary meta server M' to coordinate the rest procedure for recovery. M' will contact a remaining meta server (which might be itself) and a remaining data server (of view i) for data transferring.

Primary meta servers are key to consistency guarantees, being responsible for maintaining the latest states of objects, recovering lost data and metadata, and processing unfinished requests. To avoid inconsistent metadata when the PGs' primary meta servers change, each time a new meta server is selected, the view number is also updated. The same rule applies when a crashed meta server re-joins the system.

Lease. Cheetah uses leases [22, 27, 34] for topology map maintenance and consistency guarantees. The lease time is the length of time during which the topology map will not change. It also serves as an optimization for `get` request, allowing a `get` request to be processed by the primary meta server and any one of the n data servers. The manager renews the lease periodically, and a meta server will not answer requests if its local lease has expired. When the topology map is updated, it will become effective with the next lease.

Without leases, it is possible that a client proxy C temporarily has a stale view number and accepts the metadata (for a `get`) from an outdated meta server, which causes inconsistency. The consistency of `delete` is ensured similarly. Assuming an ongoing `delete` is launched in the current view and the meta server crashes after executing it. If the `delete` is committed in the current view by a client proxy C then the `delete` is done; otherwise C will re-issue the `delete` after topology update because the view numbers mismatch.

The manager does not wait for everyone's lease to expire, but just renews the lease periodically (before it expires). A new topology map, together with its view number, is enabled with each renewal. Upon a failure, only the failed server stops, and none of the healthy servers will be affected.

5.2 Maintaining MetaX in KV Store

Cheetah adopts a KV store (Table 1) as the local storage of meta servers for efficient MetaX maintenance and query.

First, meta servers need to quickly retrieve the metadata of given objects for efficient `get`/`delete`. Therefore, the first KV is designed to store the metadata, with key = OBMETA_*name*

Table 1. MetaX KVs.

Key	Value
OBMETA_*name*	*lvid, extents, checksum*
PGLOG_*pgid_opseq*	*name, pxlogkey*
PXLOG_*pxid_reqid*	*name, pglogkey*

and value = {*lvid, extents, checksum*}. In the key, OBMETA is a prefix for "object metadata" (for prefix matching queries), and *name* is the unique name of the object. The value includes the ID of the allocated logical volume ($\mathcal{M}_v$ = *lvid*), the allocated in-volume blocks ($\mathcal{M}_o$ = *extents*), and the object data's *checksum* (received from the client proxy).

Second, to handle a meta server (*M*) crash, for each PG on *M*, the metadata of all ongoing put of the PG must be restored by the PG's remaining meta servers, which requires recording the PGs (as meta-log). The second KV logs the PG information of the put request, with key = PGLOG_*pgid_opseq* and value = {*name, pxlogkey*}. In the key, PGLOG stands for "PG log", *pgid* is the PG's ID (= *Hash*(*name*) mod number of PGs), and *opseq* is a sequence number monotonically increased for every put in the PG. The value includes the name of the object (*name*), and the key of the client proxy log (*pxlogkey* = PXLOG_*pxid_reqid*) introduced below.

Third, to handle a client proxy (*C*) crash, the meta servers need to find *C*'s ongoing put to retrieve relevant metadata, which requires recording *C* (as meta-log). The third KV logs the client proxy (*C*), with key = PXLOG_*pxid_reqid* and value = {*name, pglogkey*}. In the key, PXLOG stands for "proxy log", *pxid* is the proxy ID, and *reqid* is the request ID generated by *C*. The value includes the unique name of the object (*name*), and the key of the PG log (*pglogkey* = PGLOG_*pgid_opseq*).

The metadata and meta-log of MetaX are maintained in the three KV pairs shown in Table 1, and updated/accessed for object put/get/delete (§4.3). For each put, The three KVs need to be atomically written, which is well supported by modern KV stores like RocksDB [3] and Badger [13].

In addition, Cheetah maintains an in-memory bitmap for each logical volume, where one bit represents the status of a block to facilitate the selection of volumes in the VG and the allocation of blocks. Each time when the VG's corresponding PG logs are cleaned, the in-memory bitmap is synchronized with its on-disk version by accumulating the new allocations (*i.e.*, *extents* in the first KV) since last synchronization.

5.3 Crash Detection and Recovery

The atomicity of MetaX precludes inconsistency between different parts of metadata, thus simplifying crash recovery.

Detection. Cheetah detects server crashes in two ways. First, client proxies will notice a potential server crash (if the server fails to respond) and report it to the manager. Second, the manager uses servers' heartbeats to detect potential failures. If a server does not respond for a period of time, it will initiate the recovery process by removing that server, updating the topology map, and disseminating the new map after the current lease expires. In addition, the primary meta server will notice a client proxy crash when processing a get for a pending object (§4.3.2) previously put by the client.

Meta server crash. If a meta server *M* becomes unavailable, Cheetah will wait for a short period of time and expect *M* to recover, marking the affected PGs as *readonly*. If *M* cannot recover in time, then the manager will consider *M* crashed and update the topology map with an increased view number. For each affected PG, based on CRUSH a new meta server will be calculated and one of the *n* meta servers will be primary. The new primary meta server cannot process any requests until the new view is synchronized to all meta servers of the PG and the data servers of the PG's VG.

A special case is that the primary meta server crashes when it is processing an ongoing put. To handle this, after the recovery completes the client proxy will resend each uncommitted put (marked as RE-META) together with the new view number to the calculated meta server, which will look up key OBMETA_*name*. If the entry does not exist, then it processes the request as a new one; otherwise it will resume Step (2)ii (§4.3.1) in the processing of the incomplete put, *i.e.*, returning the *lvid* and *extents* to the client proxy and sending the retrieved KVs to the backup meta servers, which will persist the KVs if OBMETA_*name* does not exist.

Data server crash. If a data server *D* becomes unavailable, then the manager will mark all the affected *logical* volumes as *readonly* and notify the logical volumes' primary meta servers, which will temporarily skip these volumes when processing new put. If *D* recovers in a short period, then the logical volumes will be marked as writable again and *D* will resume its service normally. Otherwise, the manager will change the topology map, select new physical volumes (from other available healthy disks) to replace the failed physical volumes of the affected logical volumes, and notifies the relevant primary meta servers about the replacement. Lost data will be restored to the new physical volumes in parallel.

A special case is that data server *D* crashes when it is writing data of an ongoing put request. Assuming the client proxy does not crash, it will re-issue the put (marked as RE-DATA) with the view number. The responsible meta server will atomically (i) select a new logical volume from the PG's VG to allocate blocks, and (ii) revoke the previous allocation on the problematic volume by clearing the bits in the bitmap.

Client proxy crash. If a client proxy *C* (ID = *pxid*) crashes, each affected primary meta server *M* (processing *C*'s put requests) will look for the proxy logs with prefix PXLOG_*pxid* which have not yet been cleaned. Each proxy log records OBMETA_*name*. *M* looks for the metadata with key OBMETA_*name*, and retrieves *lvid*, *extents* and *checksum*. Then *M* sends *lvid* and *extents* to the data servers that will return the checksums of the specified blocks. *M* compares the returned and retrieved checksums. If a checksum from *D* is

correct, then (if necessary) M will complete the put by recovering the metadata/data respectively to the unfinished meta/data servers and cleaning the logs. Otherwise, M revokes the incomplete put by deleting the KVs.

Concurrent crashes. If multiple crashes happen concurrently, Cheetah separately recovers each crash based on the coordination of the cluster manager. For instance, if both primary meta server and client proxy crash before a put is committed, then Cheetah will (i) recover the primary meta server M' with consistent MetaX records using the remaining meta servers, and (ii) let M' process the client proxy crash by waiting the timeout of uncommitted requests and checking the data servers to complete or revoke the request.

Metadata aggregation effectively prevents orphans. Suppose that the metadata server crashes before it persists the metadata, but the client proxy already gets the metadata and sends the data to the data server, and it also crashes. Then, the writes on the data server have no effect and no orphan appears, because the space allocation has not yet recorded on the meta server.

If a power loss causes all servers/clients down, after reboot the meta servers will (i) negotiate with each other for the PG logs to synchronize the MetaX KVs by comparing *opseq* in the keys, and (ii) compare *checksum* in the metadata and the checksums calculated by the n data servers on the blocks specified by *lvid* and *extents*. If none of the n checksums is correct, then the put is unfinished and will be revoked by deleting the KVs. After that the normal service can resume.

Consistency Guarantees. Cheetah provides linearizability [28, 31], *i.e.*, the effect of put/get/delete requests for the same object is equivalent to a sequential execution. Requests for different objects are irrelevant and can be executed out of order. We will prove the correctness for consistency guarantees in Appendix A.

6 Evaluation

We have implemented Cheetah based on the open-source Haystack project (named SeaweedFS) [20]. We realize Cheetah's rich meta storage (MetaX with hybrid mapping) and raw data storage to replace Haystack's thin directory storage and file-based data storage, respectively. Cheetah adopts the bitmap allocator [6] of Ceph BlueStore to manage its raw data storage. We remove distributed write ordering by enforcing local write atomicity (Fig. 1 vs. Fig. 2), so that Cheetah can perform metadata and data writes in parallel while guaranteeing linearizability.

We compare Cheetah to the directory-based object storage systems (Haystack and Tectonic). Tectonic is Facebook's general-purpose distributed filesystem that unifies the storage of hot objects (Haystack) [21], warm objects (F4) [35], and files [10], to achieve higher storage efficiency at the price of slight performance loss. Different from Haystack, Tectonic hashes objects onto log-structured files, and realizes scalable distributed file storage based on which a huge number of objects can be efficiently stored. Tectonic *disaggregates* filesystem metadata into naming, file, and block layers, each being hash-partitioned and stored in a sharded KV. It explicitly manages the mapping of chunk files to disk volumes for migration-free expansion (§2.2). For put, Tectonic first replicates the data in files and later conducts Reed-Solomon (RS) [44] encoding in batches.

For reference, we also evaluate Ceph (v17.2) [38, 49], the state-of-the-art *hash*-based object store. Ceph adopts the CRUSH algorithm to calculate the mapping of objects to volumes but suffers from data migration after expansion. Ceph uses BlueStore [7] to store object data on raw disks. We do not equip Ceph with MapX [47], as MapX can support only Ceph-RBD/Ceph-FS but not object storage. We do not test S3 [5], as S3 is an Internet-based storage service for large objects thus having poor performance for small objects.

Our testbed consists of fifteen machines. Three machines run the clients (each with a client proxy), and they also jointly run Raft as the reliable system manager. We have nine data machines and three meta machines, and adopt three-way replication for both object data and metadata. A data machine has four SSDs for object data storage (in addition to the system disk) each being divided into 100 physical volumes. Totally we have $100 \times 4 \times 9 = 3600$ physical volumes and $3600/3 = 1200$ logical volumes. We organize the logical volumes into 200 volume groups (VGs), each having six logical volumes and corresponding to one PG. Each meta machine has three SSDs for storing metadata. For meta servers, RocksDB adopts the default configurations [17].

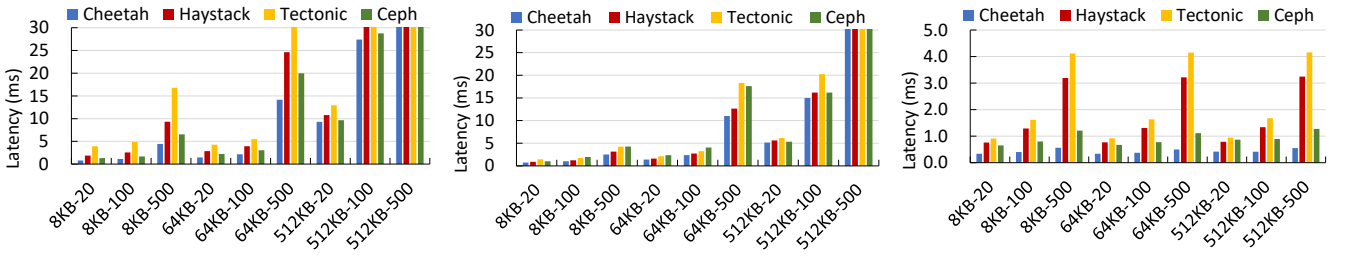
We seek to answer the following questions:

- How does Cheetah perform in micro benchmarks for latency and throughput (§6.1)?
- What are the impacts of Cheetah's various design choices, including ordering, raw block service, and RocksDB configurations (§6.2)?
- How scalable and recoverable is Cheetah (§6.3)?
- And how does Cheetah perform in trace-driven and combined workloads (§6.4)?

6.1 Micro Benchmarks

We measure the put/get/delete latencies (request completion times) of Cheetah/Haystack/Tectonic/Ceph. The clients run Intel COSBench [12] to put 10 million objects varying the number of workers (concurrency = 20/100/500). As the MetaX structure is optimal primarily for small objects, we test object data sizes of 8KB/64KB/512KB. For object data sizes $\geq$ 1MB, all systems have similar performance dominated by bulk data I/O, and thus we do not include the results in this section. The latency is directly measured by the client proxies from receiving the put to acknowledging it (Fig. 2).

The result is depicted in Fig. 5a, where, *e.g.*, 8KB-20 represents 8KB data size and concurrency = 20. For readability,



(a) put Latency **(b)** get Latency **(c)** delete Latency

Figure 5. Mean latency of Cheetah, Haystack, Tectonic and Ceph.

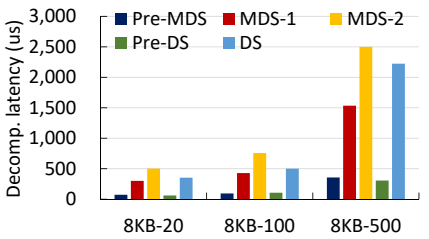


Figure 6. Latency decomposition. Y-axis uses μs, as Pre-MDS/-DS are small.

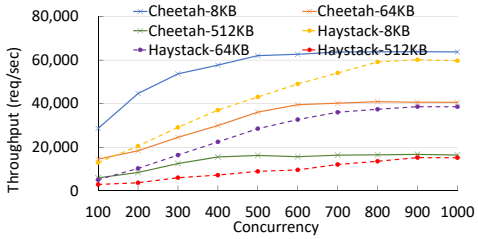


Figure 7. Throughput with different put sizes, as concurrency increases up to 1000.

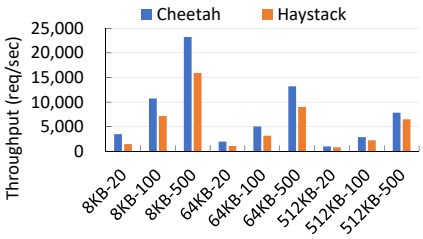


Figure 8. Performance of read-modify-write as the substitute of overwrites.

we do not show the exceptionally high latencies of large data size (512KB) with high concurrency, which are mainly affected by bulk data writes on data servers. Cheetah outperforms Haystack by up to 2.37× (at 8KB-20) in the mean put latency. This is mainly because Cheetah realizes parallel writes of metadata and data, and conducts atomic MetaX writes on meta servers avoiding separate metadata I/O. Cheetah also performs raw block I/O on data servers thus avoiding file system overhead. The latency of Cheetah is even lower than that of the hash-based Ceph, mainly because of Ceph's complex layered design affecting concurrency [31].

In addition, Ceph has to either write logs for small (≤32KB) objects or enforce *local ordering* on data servers to support the upper-layer (overwritable) Ceph-RBD and Ceph-FS. Tectonic has the highest latency, because it introduces the intermediate file abstraction while its layered filesystem metadata sharding (objects → file names → file IDs → chunk volumes) inefficiently requires multiple recursive RPCs.

We measure the get performance by performing get for 100,000 objects chosen uniformly at random from the previously put 10 million objects. The result is shown in Fig. 5b, where Cheetah outperforms Haystack by up to 25% (at 8KB-500) in the mean get latency. This is mainly because Cheetah only needs to read raw data blocks while Haystack has to read both in-volume filesystem metadata and data on data servers. Ceph has relatively high get latency, because it needs to read both metadata and data on data servers.

We measure the delete performance by removing 100,000 objects chosen uniformly at random from the previously put 10 million objects. The result is shown in Fig. 5c. Cheetah significantly outperforms Haystack in delete latency, as Cheetah accesses meta servers once to delete an object while Haystack requires multiple accesses to meta/data servers.

Cheetah significantly outperforms Haystack in put and delete latencies for both light and heavy loads (*e.g.*, 8KB-500 and 64KB-500). Therefore, Cheetah broadens the applicability of directory-based object storage to support not only read-dominant but also write-dominant applications.

Cheetah experiences latencies of at least hundreds of μs even for small (8KB) put, an order of magnitude higher than that of both network ping and pure SSD writes. As reported from industry [4], ms-level latencies for small object I/O are common even using high-end SSDs and networks. This is because an object put involves not only a network trip and disk I/O but also multiple inter-node RPCs. We decompose the overall latency of small (8KB) object put. The result is shown in Fig. 6, where Pre-MDS, MDS-1, MDS-2, Pre-DS, and DS are from the perspective of client proxies. Specifically, (i) Pre-MDS: the time of preprocessing (resolving put) and sending the request to the meta server, (ii) MDS-1: the delta time between receiving the meta server's first reply and Pre-MDS, (iii) MDS-2: the delta time between receiving the meta server's ack and MDS-1, (iv) Pre-DS: the time of sending the block write request to the data server, and (v) DS: the delta time between receiving the data server's ack and Pre-DS.

We measure the put throughput (requests per second) of Cheetah, by increasing client-side concurrency from 100 to 1000 for various object data sizes. Since Haystack always outperforms Tectonic by design, in this experiment we only compare Cheetah to Haystack. All the three client machines

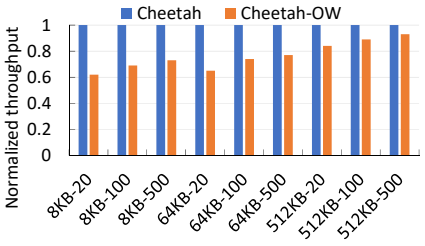


Figure 9. Impact of ordering on put.

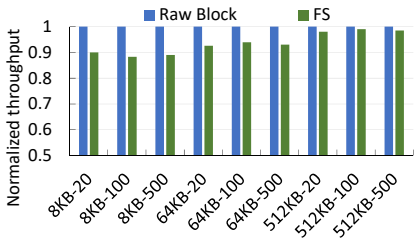


Figure 10. Impact of filesystem.

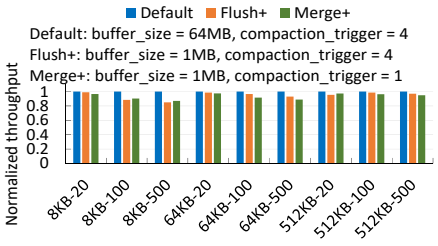


Figure 11. RocksDB configurations.

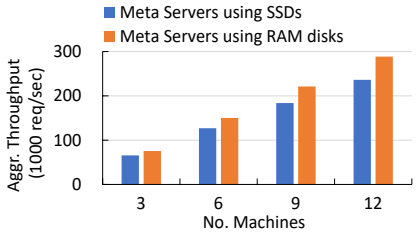


Figure 12. Meta service scalability.

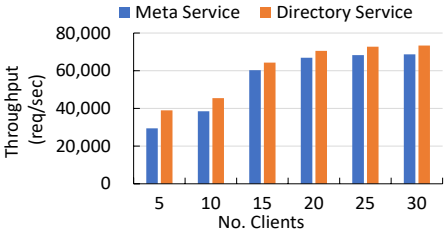


Figure 13. Meta service vs. directories.

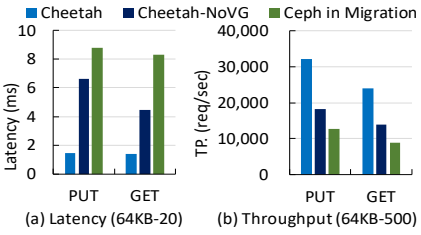


Figure 14. In-expansion performance.

and nine data server machines jointly run COSBench clients (with client proxies) to ensure clients not to be the bottleneck. The result is shown in Fig. 7.

The throughput of Cheetah is substantially higher than that of Haystack when concurrency < 600, because when the system is underloaded the throughput is a function of the concurrency and the time each put takes. After that, the advantage of Cheetah becomes smaller, and Cheetah achieves slightly higher peak throughput than Haystack (*e.g.*, about 6% improvement for 8KB-put with concurrency = 1000), as placing the offset metadata (together with the volume metadata) on meta servers better balances the load than storing it (in XFS) on data servers. We also inspect the CPU utilization for all machines. It is always < 2200%, showing that the dual 16-core CPUs are not bottleneck. Therefore, for moderate load levels, Cheetah can deliver higher throughput.

Although immutability disallows overwrites, Cheetah allows clients to delete an object and put it again with new data (§4.3.1). We write a custom benchmark tool to compare the read-modify-write performance of Cheetah and Haystack. We first put 10 million objects, and then randomly read-modify-write 100,000 of them by reading an object, deleting it, and writing it with a new value. As shown in Fig. 8, Cheetah always achieves higher throughput.

6.2 Impacts of Design Components

We evaluate the impact of parallel writes by comparing Cheetah to a variation of Cheetah with *ordered-writes* (Cheetah-OW), of which the proxies send the metadata and data to the data servers after receiving acks from the meta servers (instead of after receiving the metadata). Fig. 9 shows the normalized throughput. When the system is not saturated, Cheetah outperforms Cheetah-OW, proving the effectiveness of removing distributed ordering. This is because in Cheetah-OW the data writes must wait for their metadata writes making the pipelined write scheduling inefficient (Fig 1), while in Cheetah the metadata/data writes are in parallel (Fig. 2).

To quantify the advantage of raw block I/O, we compare the throughput of Cheetah to that of Cheetah's filesystem-backed variation (Cheetah-FS), where the data servers run XFS and use a large file on each volume to store object data. Fig. 10 shows the normalized put throughput of Cheetah-FS. The advantage of Cheetah over Cheetah-FS is relatively small (especially for 512KB objects), because the data servers of Cheetah-FS can access data of large files with low overhead. Because object I/O usually experiences millisecond-level latencies (Fig. 5), the filesystem impact on latency is negligible thus being omitted due to lack of space. For throughput, by comparing Fig. 9 and Fig. 10, we can see that for small object writes (*e.g.*, 8KB-20) the impact of the filesystem is only about 10%, while the impact of ordering can reach up to 40%. Therefore, Cheetah's performance advantage is primarily from the removal of distributed ordering.

Cheetah uses RocksDB for MetaX storage, which adopts a Log-Structured Merge-Tree [40] to store KVs. RocksDB writes the KVs both in its Memtable (of buffer_size) and in a write-ahead log. When a Memtable is full, it is *flushed* to a file and the log is cleaned. The files are organized in a sequence of levels starting from Level-0. When one level reaches its trigger, the files will be merged to the next level. By default the Memtable size is 64MB and the trigger is 4. To evaluate the impact of more aggressive flush and merge, we reduce the two values and pad the value of each KV to 1KB. Fig. 11 shows the normalized throughput compared to the original configuration. The result shows that increasing flush and merge rates only has small impact on the performance.

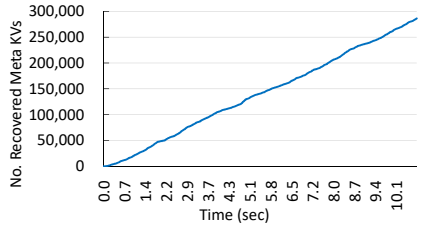


Figure 15. Meta server recovery.

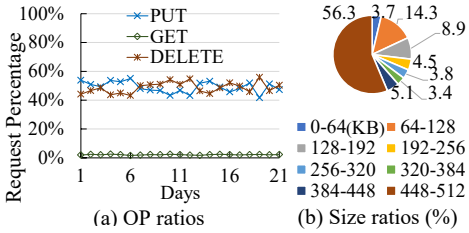


Figure 16. Op and Size ratios in the trace.

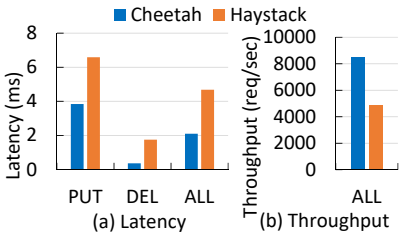


Figure 17. Trace-based comparison.

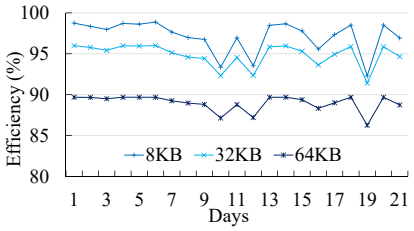


Figure 18. Storage Efficiency.

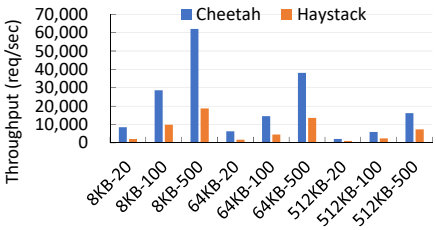


Figure 19. In-compact performance.

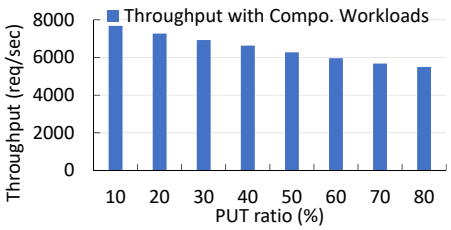


Figure 20. Combined workloads.

6.3 Meta Service Scalability and Recovery

To evaluate the scalability of the CRUSH-based meta service, we use $m = 3, 6, 9, 12$ machines to form the meta server cluster. Each of the m machine also runs a *pseudo* data server that does nothing but simply acknowledge any write requests. We use three machines to jointly run m client processes each with concurrency = 500 that can saturate the meta service. The clients put 10 million 8KB objects, and we measure the aggregate throughput of the m-machine meta service. The result is shown in Fig. 12, proving that Cheetah's meta service is scalable. For comparison we also evaluate the throughput upper bound by using RAM disks for MetaX storage.

We evaluate the impact of maintaining *all* metadata in MetaX, which introduces more load to the meta servers, on the meta service performance. To focus on the burden of meta servers, we use one single machine to run the meta service without replication, and use another machine to run the clients that bypass the processing of data servers. We also evaluate the traditional thin directory service by maintaining only the volume metadata, *i.e.*, storing one KV (*name* → allocated logical volume) in RocksDB. We compare the throughput of the meta and directory services, as the number of client processes increases. As shown in Fig. 13, the performance of the rich meta service is slightly lower than that of the simple directory service. Compared to traditional directory-based object stores, Cheetah can adjust the number of meta/data servers, as it moves the maintenance load of offset metadata from data servers to meta servers.

We evaluate the migration-free feature of Cheetah, with the same configuration as in §6.1 except that we use eight data machines and reserve the ninth for *meta* service expansion. We first put 10 million random-sized objects to Cheetah, and then add the reserved machine to the CRUSH-based *meta* service cluster. CRUSH remaps 1/4 of the PGs to the new meta machine, after which we measure the put/get performance of Cheetah (with VG) and Cheetah-NoVG (without VG). As depicted in Fig. 14, Cheetah performs much better as Cheetah-NoVG suffers from data migration. We further test in-expansion Ceph that adopts CRUSH to map objects onto disk volumes. After adding one data server, the performance of Ceph degrades due to the contention of data migration. In contrast, Cheetah is affected neither by meta server expansion nor by data server expansion.

We test the recovery of a meta server crash. We (i) write 8KB objects with concurrency = 100 for 10 sec, (ii) disconnect one of the three meta machines, and (iii) re-connect it as a new one. After detecting the crash, the manager will disseminate the new topology map. The metadata of the affected PGs will be recovered to the new meta servers. As shown in Fig. 15, all metadata is recovered within a few seconds.

We also test the recovery of a disk failure in Cheetah and Ceph. We first put 10 million 512KB objects, having each disk store 512KB × 10 million × 3 / 36 ≈ 406.9GB data. Cheetah takes about 16.3 sec to recover the lost data to healthy disks in parallel with an aggregate throughput of 24.9 GB/sec. In contrast, Ceph takes about 16.1 sec for recovery, which is slightly faster owing to its CRUSH-based data placement.

6.4 Traces and Combined Workloads

We collect a three-week trace of the workload of 24 storage machines in production, which stores data as objects for various analysis applications of a large-scale Internet service. Fig. 16a shows the ratios of the numbers of different requests for each day, where (i) there are much more writes (put) than reads (get), and (ii) the ratio of delete is high because most objects have a lifecycle (ranging from hours to months). The object size ratios are shown in Fig. 16b. We

use a custom benchmark tool (ignoring the timestamps) to replay the recorded I/O requests. We evaluate the latency and throughput. The results (Fig. 17) show that Cheetah outperforms Haystack for production workloads, where there are both large and small objects and some requests can be cached due to locality.

Cheetah's raw data storage allows direct reuse of the reclaimed space. Cheetah can also resort to compaction if necessary. We evaluate Cheetah's storage efficiency, *i.e.*, the total object data size divided by the total capacity occupied on data servers, at the end of each hour during the replay of the three-week trace. Fig. 18 shows the maximum efficiency for each day, where Cheetah always achieves high efficiency (> 85%). The drops are because of scheduled batch deletes.

We evaluate the impact of Haystack's compaction. We fill the system with objects and trigger a compaction by randomly deleting objects. We put no limitations on the bandwidth used by compaction, and measure the in-compaction `put` throughput. The result is shown in Fig. 19, which also includes (compaction-free) Cheetah. The advantage of Cheetah gets much higher when Haystack is in compaction, and thus Cheetah is more suitable when deletions are common for objects with unpredictable lifecycles. Note that if we limited the compaction bandwidth, although Haystack's in-compaction degradation would be slighter, Haystack would experience less severe but longer performance degradation as compaction inevitably causes amplification, *i.e.*, all objects in the old files must eventually be moved to the new ones.

We use YCSB [11] to generate combined workloads. The `delete` ratio is 10%, the `put` ratio varies from 10% to 80%. Object sizes are chosen randomly between 4 ∼ 512 KB. We run the workloads with concurrency = 20. As shown in Fig. 20, The throughput slightly decreases as the `put` ratio increases, proving Cheetah is applicable to a wide range of workloads.

7 Discussion

Rich metadata. Cheetah follows the technical trend of breaking machine boundaries, aggregating metadata into MetaX to realize separate metadata/data storage. This design choice potentially has two drawbacks. First, it increases meta servers' complexity. To address this problem, we propose hybrid mapping to achieve high scalability (§4.2). Second, it makes the data servers not self-contained which may hurt data durability. To address this problem, we carefully design the recovery mechanism to guarantee the correctness including data durability (§5). For instance, if multiple components (like the meta server and the client proxy) simultaneously crash, the writes on data servers have no effect and no orphans occur, as the space allocation (to which data servers are agnostic) has not yet recorded on the meta server.

The offset metadata needs to be maintained on either meta servers or data servers. Cheetah moves it from data servers to meta servers (in the aggregated MetaX), to remove distributed ordering and enable parallel metadata/data writes. Experimental results of Figs. 5∼8 (all using three machines for MDS) show the latency and throughput benefits. Fig. 12 shows that Cheetah's hybrid mapping alows *linear scale-out* to many meta servers. Therefore, Cheetah's MDS is scalable. Fig. 13 shows that RocksDB can atomically write a batch of KVs with little performance overhead, and thus the increased metadata complexity is not a problem for scalability.

Read optimization. Similar to Haystack, Cheetah can optimize `get` if reading an object previously put by the same client proxy. Specifically, C caches the metadata ($\mathcal{M}_v$ = *lvid*, $\mathcal{M}_o$ = *extents*) before acknowledging a `put` for subsequent `get` requests. When receiving a cache-hit `get`, C will perform Step (2) and Steps (3)(4) in parallel (§4.3.2). Optionally, it can use the checksum from M_1 to detect tampered $\mathcal{M}_v$ and $\mathcal{M}_o$.

Availability. Cheetah makes use of the standard synchronous replication scheme for high data durability. A write request is committed only if its metadata is replicated by f meta servers and object data is replicated by f data servers. A potential inefficiency is the necessity of synchronous failure recovery, during which the write procedure of corresponding VGs and PGs has to be suspended thus affecting availability. For higher availability, Cheetah can (slightly) trade off durability by leveraging asynchronous or hybrid replication [31, 48]. This is orthogonal to the main design of this paper and will be studied in our future work.

Immutability. Cheetah leverages object immutability to remove distributed write ordering and simplify data maintenance and message exchange, while still guaranteeing consistency. Object immutability needs to be guaranteed by upper-layer applications (like Facebook's photo-uploader) generating unique object names. Without immutability, for Cheetah it would be possible that a misbehaved user `put`'s metadata overwrites an existing object but that `put`'s data is lost (*i.e.*, metadata-data inconsistency). To address this problem, Cheetah can generate a unique sub-name for each `put` to keep both versions' metadata until the `put` is committed. In addition, MetaX can also support overwrites by adopting two-phase commit [33], which inevitably lowers the object write performance. Two-phase commit is necessary for application scenarios of mutable objects (like Ceph-RBD and Ceph-FS). The main difference lies in the metadata management, as data servers are agnostic to the status of objects.

Directories vs. hashing. The choice between directory-based and hash-based object mapping is a dilemma for fast-expanding businesses with ever-increasing storage demands. On one hand, hash-based mapping provides high scalability and performance through decentralized and consistent calculation for object placement in serving normal object I/O requests, but causes severe data migration in capacity expansion. For instance, when adding one rack to a three-rack Ceph cluster, almost 60% of the PGs will be affected [47], which will inevitably cause performance degradation during

the entire migration period. On the other hand, directory-based mapping achieves migration-free expansion by maintaining and querying object placement with a centralized directory, but suffers from poor performance for small object I/O. For instance, when we adopt Haystack as the underlying object store for our businesses, as the total data volume increases from PBs (petabytes) to EBs (exabytes), the centralized directory service becomes a significant bottleneck when numerous clients from different collection/analysis applications (§6.4) access object storage in parallel. This motivates us to propose the hybrid mapping architecture (§4.2), which realize the scalable (meta) directory service without expansion-caused data migration. Further, we find that even though we eliminate the centralized directory bottleneck, applications still experience unsatisfactory performance especially in small I/O latency, which is because of the distributed ordering constraint on the object I/O path.

8 Conclusion

In existing directory-based object stores, it is the *separate metadata maintenance* that complicates object write operations and enforces distributed write ordering, which consequently lower the performance of small object I/O. Our insight is that all metadata of an object can be aggregated into MetaX, so that we can replace the distributed ordering constraint with the much simpler local write atomicity of MetaX. We leverage MetaX to implement an efficient object store called Cheetah, which significantly improves I/O performance of `put`/`get`/`delete` for immutable objects while still ensuring consistency. In our future work, we will integrate Cheetah with erasure coding [32] for high efficiency and asynchronous replication [46, 48] for high availability.

Acknowledgment

We would like to thank our shepherd, Shuai Mu, and the anonymous reviewers for their insightful comments. Yiming Zhang and Li Wang are the co-primary authors. We thank Chunhua Huang and Mingya Shi for an early version of the Cheetah implementation. The work is supported by the National Key Research and Development Program of China (grant no. 2022YFB4500302) and the National Natural Science Foundation of China (grant no. 62025208). Yiming Zhang is the corresponding author.

Appendix

Appendix A: Proof of Correctness

We prove that Cheetah ensures the following invariants, which together guarantee consistency.

Lemma 1. *If a* `put` *request is committed and the primary meta server is notified, then all subsequent* `get` *requests will see the data unless a* `delete` *is committed thereafter.*

Proof. Different from existing work, Cheetah writes metadata and data of an object in parallel without distributed ordering. Therefore, we focus on the argument that this parallel processing does not affect the correctness.

It is trivial to see that all future get will see the data within the same view. Now we argue that Lemma 1 holds after updating the topology map, even if Cheetah writes metadata and data in parallel. Suppose that an ongoing `put` request, req_p, is issued (but may not be committed) in view i, and that $f = n-1$ meta severs storing the metadata of req_p and f data servers storing the data of req_p either crash permanently or are partitioned due to network issues. The remaining meta and data servers that are updated to view $i+1$ are M and D, respectively.

We discuss two cases. First, if req_p is already committed by the proxy in view i, then both M and D should have written the metadata and object data of req_p in view i. Owing to immutability, the metadata of req_p will not be replaced by a new `put` at M, and thus M and D will relay the metadata/data of req_p to the new servers in view $i+1$. The object can be accessed after recovery. Note that a `delete` request does not introduce distributed write operations owing to the MetaX structure. Either a `delete` succeeds in view i after req_p and M should not store the metadata of req_p anymore; or req_p can be recovered in view $i+1$ with the help of M, which is also acceptable.

Second, if the metadata of req_p is not written on M or the object data of req_p is not written on D, then the proxy does not commit req_p in view i and will re-issue the request in view $i+1$ once it is notified about the topology and view updates. If the metadata of req_p is written on M, but D does not have the data, then M will abort req_p and revoke its processing since the checksum does not match. In this case the proxy cannot commit req_p in view i anymore, because either M or D will reject req_p due to the mismatching of view numbers (req_p is in view i, but M or D is already in view $i+1$). Therefore, Cheetah guarantees that either req_p is committed in view i and will be durable since then, or req_p is not committed at all.

Finally, suppose that a `get` request (req_g) for the same object is issued after the topology map update. If req_g is with view i then it will not be served by any servers because of the mismatching of view numbers and the lease mechanism. So req_g must be re-issued in view $i+1$, and thus the corresponding meta servers and data servers in view $i+1$ can reply with correct metadata and data, respectively. □

Lemma 2. *If a* `get` *request sees an object, then all subsequent* `get` *requests will see the same data unless a* `delete` *request is committed thereafter.*

Proof. A `get` request sees an object only if the corresponding `put` request is committed and the client proxy has notified the primary meta server (Line 10 in Algorithm 1). According to Lemma 1, all future `get` requests will see the same data. □

References

[1] 2012. Twitter Photo Storage. https://blog.twitter.com/engineering/en_us/a/2012/blobstore-twitter-s-in-house-photo-storage-system.html. Accessed: 2024-05-02.

[2] 2022. InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 313–328. https://www.usenix.org/conference/fast22/presentation/lv

[3] 2023. A Persistent Key-Value Store for Fast Storage Environments. https://rocksdb.org/. Accessed: 2024-05-02.

[4] 2023. All-NVMe Performance Deep Dive Into Ceph. https://flashmemorysummit.com/English/Collaterals/Proceedings/2018/20180807_INVT-101A-1_Meredith.pdf. Accessed: 2024-05-02.

[5] 2023. Amazon Simple Storage Service (Amazon S3). https://aws.amazon.com/s3/. Accessed: 2024-05-02.

[6] 2023. Ceph Bitmap Allocator. https://github.com/ceph/ceph/blob/master/src/os/bluestore/Allocator.cc. Accessed: 2024-05-02.

[7] 2023. Ceph BlueStore. https://ceph.io/community/new-luminous-bluestore/. Accessed: 2024-05-02.

[8] 2023. Ceph RBD. https://ceph.com/ceph-storage/block-storage/. Accessed: 2024-05-02.

[9] 2023. Facebook. https://www.facebook.com/. Accessed: 2024-05-02.

[10] 2023. HDFS Architecture Guide. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html. Accessed: 2024-05-02.

[11] 2023. Implementing New Workloads. https://github.com/brianfrankcooper/YCSB/wiki/Implementing-New-Workloads. Accessed: 2024-05-02.

[12] 2023. Intel COSBench: Cloud Object Storage Benchmark. https://github.com/intel-cloud/cosbench. Accessed: 2024-05-02.

[13] 2023. Introducing Badger: A Fast Key-Value Store Written Purely in Go. https://dgraph.io/blog/post/badger/. Accessed: 2024-05-02.

[14] 2023. LinkedIn Object Storage. https://www.linkedin.com/. Accessed: 2024-05-02.

[15] 2023. Multi-Cloud Object Storage. https://min.io/. Accessed: 2024-05-02.

[16] 2023. OpenStack Object Storage (Swift). https://wiki.openstack.org/wiki/Swift. Accessed: 2024-05-02.

[17] 2023. RocksDB Setup Options and Basic Tuning. https://github.com/facebook/rocksdb/wiki/Setup-Options-and-Basic-Tuning. Accessed: 2024-05-02.

[18] 2023. SSD Prices Have Dropped 25% Since March, Now Average $0.06 per GB. https://www.tomshardware.com/news/ssd-prices-sink-june-2023. Accessed: 2024-05-02.

[19] 2023. SSDs will crush hard drives in the enterprise, bearing down the full weight of Wright's Law. https://blocksandfiles.com/2021/01/25/wikibon-ssds-vs-hard-drives-wrights-law/. Accessed: 2024-05-02.

[20] 2024. Seaweedfs. https://github.com/seaweedfs/seaweedfs. Accessed: 2024-05-02.

[21] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, and Peter Vajgel. 2010. Finding a needle in Haystack: facebook's photo storage. In *Usenix Conference on Operating Systems Design and Implementation*. 47–60.

[22] Mike Burrows. 2006. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06)*. USENIX Association, Berkeley, CA, USA, 335–350. http://dl.acm.org/citation.cfm?id=1298455.1298487

[23] 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*. ACM, 228–243.

[24] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. Consistency without ordering. In *Usenix Conference on File and Storage Technologies*.

[25] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (April 1985), 374–382. https://doi.org/10.1145/3149.214121

[26] C. Gray and D. Cheriton. 1989. Leases: An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP '89)*. ACM, New York, NY, USA, 202–210. https://doi.org/10.1145/74850.74870

[27] C. Gray and D. Cheriton. 1989. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. *SIGOPS Oper. Syst. Rev.* 23, 5 (Nov. 1989), 202–210. https://doi.org/10.1145/74851.74870

[28] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

[29] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. 1997. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *Distributed Algorithms*, Marios Mavronicolas and Philippas Tsigas (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 126–140.

[30] Haoyuan Li. 2018. *Alluxio: A Virtual Distributed File System*. Ph. D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-29.html

[31] Huiba Li, Yiming Zhang, Dongsheng Li, Zhiming Zhang, Shengyun Liu, Peng Huang, Zheng Qin, Kai Chen, and Yongqiang Xiong. 2019. Ursa: Hybrid Block Storage for Cloud-Scale Virtual Disks. In *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, 15.

[32] Huiba Li, Yiming Zhang, Zhiming Zhang, Shengyun Liu, Dongsheng Li, Xiaohui Liu, and Yuxing Peng. 2017. PARIX: speculative partial writes in erasure-coded systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, 581–587.

[33] C. Mohan, B. Lindsay, and R. Obermarck. 1986. Transaction Management in the R* Distributed Database Management System. *ACM Trans. Database Syst.* 11, 4 (Dec. 1986), 378–396. https://doi.org/10.1145/7239.7266

[34] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2014. Paxos Quorum Leases: Fast Reads Without Sacrificing Writes. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/2670979.2671001

[35] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, and Linpeng Tang. 2014. f4: Facebook's warm BLOB storage system. In *Usenix Conference on Operating Systems Design and Implementation*. 383–398.

[36] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. 2017. HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 89–104. https://www.usenix.org/conference/fast17/technical-sessions/presentation/niazi

[37] Shadi A. Noghabi, Sriram Subramanian, Priyesh Narayanan, Sivabalan Narayanan, Gopalakrishna Holla, Mammad Zadeh, Tianwei Li, Indranil Gupta, and Roy H. Campbell. 2016. Ambry:LinkedIn's Scalable Geo-Distributed Object Store. In *International Conference on Management of Data*. 253–265.

[38] Myoungwon Oh, Sungmin Lee, Samuel Just, Young Jin Yu, Duck-Ho Bae, Sage Weil, Sangyeun Cho, and Heon Y Yeom. 2023. TiDedup: A New Distributed Deduplication Architecture for Ceph. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 117–131.

[39] Brian M. Oki and Barbara H. Liskov. 1988. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing* (Toronto, Ontario, Canada) *(PODC '88)*. ACM, New York, NY, USA, 8–17. https://doi.org/10.1145/62546.

62549

[40] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.* 33, 4 (1996), 351–385. https://doi.org/10.1007/s002360050048

[41] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIXATC 14)*. 305–319.

[42] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. 2021. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 217–231.

[43] Raghu Ramakrishnan, Baskar Sridharan, John R. Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, Neil Sharman, Zee Xu, Youssef Barakat, Chris Douglas, Richard Draves, Shrikant S. Naidu, Shankar Shastry, Atul Sikaria, Simon Sun, and Ramarathnam Venkatesan. 2017. Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 51–63. https://doi.org/10.1145/3035918.3056100

[44] I. S. Reed and G. Solomon. 1960. Polynomial Codes Over Certain Finite Fields. *J. Soc. Indust. Appl. Math.* 8, 2 (1960), 300–304. https://doi.org/10.1137/0108018

[45] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review* 31, 4 (2001), 149–160.

[46] Robbert van Renesse and Fred B. Schneider. 2004. Chain Replication for Supporting High Throughput and Availability. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, Eric A. Brewer and Peter Chen (Eds.). USENIX Association, 91–104. http://www.usenix.org/events/osdi04/tech/renesse.html

[47] Li Wang, Yiming Zhang, Jiawei Xu, and Guangtao Xue. 2020. MAPX: Controlled Data Migration in the Expansion of Decentralized Object-Based Storage Systems. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*.

[48] Yang Wang, Lorenzo Alvisi, and Mike Dahlin. 2012. Gnothi: Separating Data and Metadata for Efficient and Available Storage Replication. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 38–38. http://dl.acm.org/citation.cfm?id=2342821.2342859

[49] 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.

[50] 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.