Cheetah: High-Throughput and Low-Latency Object Storage Based on the Write-Optimal METAX Structure

Paper #110, 12 pages

Abstract

Commercial object stores usually rely on directories for object-to-volume mapping (referred to as volume meta-data), so as to support efficient system management such as migration-free expansion and flexible auditing. However, the separation between volume meta-data (in directories) and filesystem meta-data (on data servers) severely affects object write performance: in order to guarantee crash consistency, for each `put` request multiple distributed writes have to be orchestrated in a particular *order*, respectively for the volume meta-data, filesystem meta-data and data, and `put` meta-log.

To address this problem, in this paper we design an object-write-optimal structure (called METAX) that contains all meta information (meta-data/-log) of an ongoing `put`. We then replace the expensive *distributed ordering* constraint with the (much simpler) *local atomicity* requirement of writing METAX on meta servers, which can be parallelized with writing object data on data servers while still preserving crash consistency of *immutable* objects. We further implement an efficient object store called *Cheetah*, which integrates METAX with novel designs for high performance, scalability, reliability, and availability. Extensive evaluation shows that Cheetah significantly outperforms state-of-the-art object stores (Haystack and Ceph) in object I/O performance.

1 Introduction

Object storage has been widely adopted in OLDI (online-data-intensive) applications of which huge volumes of online data are stored as large numbers of relatively-small objects. Object storage provides a simple interface of `put`/`get`/`delete` for writing/reading/deleting objects, but does not allow overwrites of data of existing objects (a.k.a. *immutability* [55]).

The mapping from objects to disk volumes is critical for I/O performance of large-scale object storage. There are two categories of methods for object-to-volume mapping, namely, directory-based volume meta-data maintenance [26, 55] and hash-based volume calculation [66, 67]. Commercial object stores with fast-expanding business (such as Facebook [23] and LinkedIn [24]) usually prefer directory-based methods to hash-based ones, so as to support efficient system management such as migration-free expansion and flexible auditing. For instance, Haystack [26] uses directories to flexibly place objects onto appropriate volumes, avoiding data migration [64] after capacity expansions which are common for OLDI applications with rapidly-growing storage demand.

However, storing volume meta-data in directories greatly complicates the design for crash consistency, since crashes may happen during the separate writes respectively in directories and on storage data servers. In order to ensure crash consistency for a `put`, multiple distributed write operations have to be orchestrated in a particular *order* (§2), respectively for the volume meta-data, filesystem meta-data and data, and `put` meta-log. Enforcing such ordering severely affects object write performance, especially for modern object stores which use fast SSDs (solid state drives) for storage of not only volume meta-data but also filesystem meta-data and data to achieve high object I/O performance.

To address this problem, in this paper we design an object-write-optimal structure called METAX (§3) that contains all meta information (meta-data/-log) of an ongoing `put`. We then replace the expensive *distributed ordering* constraint with the (much simpler) *local atomicity* requirement of writing METAX on meta servers, which can be parallelized with writing object data on data servers while still preserving crash consistency of *immutable* objects. We further implement Cheetah object store, which integrates METAX with designs for high performance, scalability, reliability, and availability.

First, we propose a novel hybrid mapping architecture (§4) for scalable METAX storage where the objects are first mapped onto the meta servers via calculation (CRUSH [67]) and then allocated to the volumes of data servers by the meta servers, so as to achieve both migration-free expansion of data storage and high scalability of meta-data storage. Second, we leverage METAX to realize efficient object I/O (§5), which (i) accelerates `put` and `get` by directly accessing in-volume raw data blocks on data servers without filesystem overhead, and

(ii) effectively reduces *compaction* [26] for `delete` compared to file-based object data storage by minimizing deletion-caused fragmentation. Third, we design Cheetah's recovery mechanism (§6) which uses lease [38] and viewstamp [57] to provide high reliability and availability, and prove that Cheetah guarantees consistency under various failures.

To the best of our knowledge, Cheetah is the first directory-based object store that ensures crash consistency without distributed write ordering. We compare Cheetah to (the latest versions of) the state-of-the-art object stores, Haystack [26] and Ceph [66], using both micro benchmarks and trace-driven experiments. Extensive evaluation shows that Cheetah significantly outperforms them in object I/O performance.

2 Background

2.1 Object Placement

An object usually has a unique name, actual data, and optionally some attributes. Large-scale object storage places objects onto appropriate disk volumes either by adopting directory-based volume meta-data maintenance or by using hash-based volume calculation. Hash-based methods usually provide better performance than directory-based ones, since they directly calculate the target volumes without accessing intermediate directories. CRUSH [67] is the state-of-the-art hash-based placement method. It models the cluster topology as a logical tree and introduces *placement groups* (PGs) to facilitate the calculation of the placement: an object is first mapped to a PG by calculating its PGID = $Hash(name)$ mod *number of PGs*; and then a PG is mapped onto multiple volumes by executing specific rules top-down in the tree.

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. E.g., almost 60% of the PGs will be affected when adding one rack to a three-rack CRUSH-based cluster [64]. Although this problem could be temporarily mitigated by limiting the migration rate [66], all objects still have to be *eventually* placed onto the calculated volumes, resulting in an even longer period of degradation. Ring-based hashing [62] suffers from similar problem of *uncontrollable* migration with CRUSH, but is much less flexible in modeling the hierarchy of racks, machines, disks, etc. MapX [64] uses time-dimension mapping (from object creation times to expansion times) for object-based *block* storage [4] to avoid such migration. However, it cannot store common objects due to the overwhelming per-object timestamp overhead.

Compared to hash-based placement, directory-based placement can easily realize migration-free expansions by managing the mapping from objects to volumes, keeping existing objects unaffected after expansions and placing new objects onto new volumes. Further, directory-based placement enables flexible system management [26, 55] such as auditing, remapping and selectively archiving less popular data.

2.2 Challenge of Directory-Based Placement

In directory-based object stores, a central directory service maintains the mapping from objects to volumes referred to as *volume meta-data* (a.k.a. application meta-data in [26]). To reduce the per-object meta-data overhead suffered by small-file-based object stores (like Swift [20] and S3 [2]) where each object is stored as a small file, a common optimization widely adopted by modern object stores [26, 55] is to append objects to volume files, which requires to maintain the local mapping from an object to its file offset (i.e., *filesystem meta-data*).

The separation of the volume meta-data (in directories) and the filesystem meta-data and object data (on data servers) brings potential risks of inconsistency. When a client `put` an object, the meta-data and data need to be consistently updated [32]: (i) all distributed meta-data should be consistent with each other (i.e., meta-data consistency), and (ii) the meta-data should be consistent with its data (i.e., data consistency).[1]

Currently, directory-based object stores [26, 55] rely on distributed ordering to maintain consistency in the face of crashes. Such ordering introduces waiting into the critical path of writing objects and constrains I/O scheduling both at the disk level and at the network level, thus drastically lowering the performance. The root cause of the necessity of ordering is that the separation of the meta information greatly complicates the reasoning about which was already persisted and which was not in the event of a crash, which is essential for the system to take corrective measures to recover.

2.3 Write Ordering for Crash Consistency

This subsection discusses distributed write ordering for crash consistency. We take Facebook's Haystack object store (for photo sharing [26]) as a representative, and other directory-based object stores [15, 55] have similar ordering constraints.

Fig. 1 shows the processing of a `put` request in Haystack, where the storage system consists of a proxy server cluster, a directory server cluster, and a data server cluster. For simplicity of discussion, we omit object attributes and assume one-way replication where each object has only one replica so that it could prevent data loss from temporary crashes and power failures but not from permanent disk/machine crashes. Besides, object immutability simplifies the ordering relationship between *multiple* `put` requests because a later `put` will not overwrite an existing object, and thus we could focus only on the ordering between the multiple write operations for an *individual* `put`. To ensure meta-data and data consistency [32] under various proxy/directory/data server crashes, the processing in Haystack is as follows.

(1) The proxy server P receives a `put` request. P first logs the object name ($name$), selected directory server (Q), and checksum (c), and then sends the name and size to Q. We refer to the logging operation as $\mathcal{L}_P = \mathcal{L}(name, Q, c)$.

[1]There is a third kind of *version consistency* (for the meta-data version to match the data version) [32], which is negligible for immutable objects.

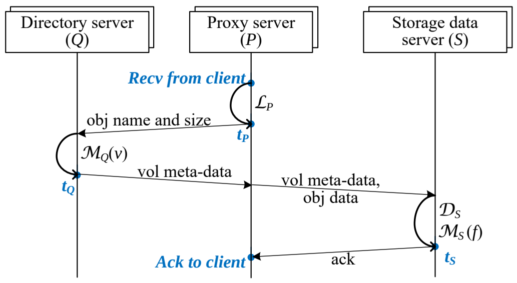


Figure 1: Distributed write ordering. t_P, t_Q, t_S represents the times when the disk writes on P, Q, S are done, respectively.

(2) The directory server Q allocates a volume (i.e., volume meta-data denoted as $\mathcal{M}(v)$) for the object, persists (i.e., locally writes) the volume meta-data, and returns $\mathcal{M}(v)$ to P. Note that Q cannot return $\mathcal{M}(v)$ before its persistence. Otherwise if the object data is successfully persisted on the storage data server (S) and both P and Q fail before $\mathcal{M}(v)$ is persisted, then the persisted data will become orphan. We refer to the write of $\mathcal{M}(v)$ on Q as $\mathcal{M}_Q(v)$.

(3) The proxy server P sends the object data ($\mathcal{D}$) and the volume meta-data ($\mathcal{M}(v)$) to the responsible data server S.

(4) The data server S allocates space in the specified volume (i.e., filesystem meta-data $\mathcal{M}(f)$), and persists $\mathcal{D}$ and $\mathcal{M}(f)$ (denoted as $\mathcal{D}_S$ and $\mathcal{M}_S(f)$). It then returns to P, which further returns to the client after verification. At this point the log $\mathcal{L}_P$ can be cleaned and the `put` is said to be *committed*.

The following sequence of distributed writes (each being denoted by a symbol "ς" in Fig. 1) take place in order: $\mathcal{L}_P$ before $\mathcal{M}_Q(v)$ before $\mathcal{D}_S$ and $\mathcal{M}_S(f)$, or more simply $\mathcal{L}_P \rightarrow \mathcal{M}_Q(v) \rightarrow \mathcal{D}_S \odot \mathcal{M}_S(f)$, where $\rightarrow$ represents the *distributed ordering* relationship, and $\odot$ represents the *local atomicity* (or *transaction* [31]) relationship of multiple writes on a single node. Since local atomicity has been extensively studied [32] and well supported based on local ordering and logs for $\mathcal{D}_S$ and $\mathcal{M}_S(f)$ (which fall out of the scope of this paper), next we will focus only on the distributed ordering relationship.

Consider a power failure that possibly takes place at any time t and temporarily fails any servers. The first order $\mathcal{L}_P \rightarrow \mathcal{M}_Q(v)$ enables P to track uncommitted `put` requests: if a failure happens at $t > t_P$, then P can check whether a `put` logged in $\mathcal{L}_P(n)$ is finished by first querying Q with *name* and then querying S with $\mathcal{M}(v)$. The second order $\mathcal{M}_Q(v) \rightarrow \mathcal{D}_S \odot \mathcal{M}_S(f)$ ensures that persisted data on S will not become orphan due to a failure at time $t > t_S$ because $t_S > t_Q$. Note that $\mathcal{M}_Q(v)$ can be safely written without worrying that a failure at $t_Q < t < t_S$ would both fail the current `put` and invalidate the meta-data of an existing object with the same *name*, because Haystack's photo uploader guarantees immutability.

Early object stores adopted slow HDDs (hard disk drives) for object data storage, and they could alleviate the impact of distributed ordering by replacing slow HDDs with fast SSDs on directory servers to reduce the delay of writing volume meta-data ($\mathcal{M}(v)$) compared to that of writing filesystem meta-data ($\mathcal{M}(f)$) and data ($\mathcal{D}$). For high I/O performance, however, modern object stores use SSDs for storage of not only $\mathcal{M}(v)$ but also $\mathcal{M}(f)$ and $\mathcal{D}$, making the ordering constraint a critical factor limiting object write performance.

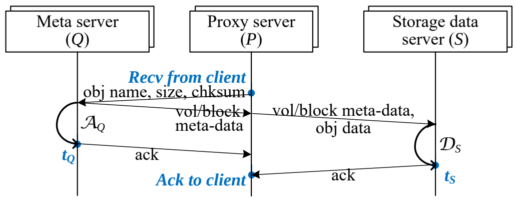


Figure 2: Parallel writes of meta-data/data without ordering.

3 The METAX Structure

3.1 Consistency without Distributed Ordering

Since the root cause of distributed ordering is the separately-maintained meta information, our key idea is to place all the volume/filesystem meta-data and meta-log into a single data structure (METAX), which can be atomically written with high efficiency by existing local filesystems [26, 31, 32] or key-value (KV) stores [7, 19, 59]. Compared to traditional directory-based methods (Fig. 1), we extend the *thin directory* service to the *rich meta* service for METAX storage (Fig. 2). Accordingly, the proxy servers serve as a *stateless* surrogate that forwards requests and returns to the clients; and the data servers provide *raw* block service that is agnostic to objects and simply reads/writes data blocks specified by meta service.

The rich meta service is responsible for maintaining (i) the mapping from objects to volumes, i.e., *volume meta-data* $\mathcal{M}(v)$, (ii) the mapping from objects to blocks, which we refer to as *block meta-data* $\mathcal{M}(b)$, and (iii) the meta-log including the object name (*name*), proxy server (P), and data checksum (c), denoted as $\mathcal{L} = \mathcal{L}(name, P, c)$. We refer to the set of all meta information (including $\mathcal{M}(v)$, $\mathcal{M}(b)$, and $\mathcal{L}$) as METAX ($\mathcal{A}$), and denote the write of METAX on meta server Q as $\mathcal{A}_Q = \mathcal{M}_Q(v) \odot \mathcal{M}_Q(b) \odot \mathcal{L}_Q$. Next we discuss how to remove the distributed ordering constraint while still ensuring consistency by atomically persisting $\mathcal{A}_Q$.

3.2 Writing METAX with Local Atomicity

By integrating all meta-data/-log into METAX, the separate writes on meta servers and data servers can be processed in parallel, as shown in Fig. 2 (adopting one-way replication).

(1) The proxy server P sends the object name (*name*), size and checksum (c) to a meta server Q.

(2) The meta server Q allocates the volume ($\mathcal{M}(v)$) and blocks ($\mathcal{M}(b)$) for object data, and sends $\mathcal{M}(v)$ and $\mathcal{M}(b)$ to P. It then persists $\mathcal{A}_Q = \mathcal{M}_Q(v) \odot \mathcal{M}_Q(b) \odot \mathcal{L}_Q$ and returns.

(3) The proxy server P sends the object data as well as $\mathcal{M}(v)$ and $\mathcal{M}(b)$ to the corresponding data server S.

(4) The data server S persists the data ($\mathcal{D}_S$) according to $\mathcal{M}(v)$ and $\mathcal{M}(b)$. It then acknowledges with the checksum

to the proxy server P, which verifies the checksum and acknowledges to the client (after receiving acks from both Q and S). At this point of time the `put` request is *committed*. P asynchronously notifies Q to clean the logs.

Clearly, there is no ordering constraint for the two separate writes ($\mathcal{A}_Q$ and $\mathcal{D}_S$). Next, we briefly discuss how the atomicity of $\mathcal{A}_Q$ guarantees consistency without ordering. Suppose that a power failure takes place and can temporarily fail any servers. Since the meta server Q maintains all meta information that could be used for recovery, we analyze the consistency by considering the following two cases of Q.

First, suppose that the proxy server P and/or the data server S temporarily crash but the meta server Q keeps alive. After recovery, for each meta-log ($\mathcal{L}$) Q queries the corresponding data server S with $\mathcal{M}(v)$ and $\mathcal{M}(b)$ which will compute and return the checksum (c) to Q. It then verifies c, and notifies P the `put` is complete if correct. If incorrect, Q will ask P to re-send the data ($\mathcal{D}$) if P has $\mathcal{D}$ or simply abort otherwise.

Second, suppose that the meta server Q temporarily crashes. If the proxy server P already receives all acks from Q then P simply ignores it. Otherwise, (after recovery) P re-sends the name, size, and c of each unacknowledged request to Q. The recovered meta server Q will either (i) return the previously-assigned meta-data ($\mathcal{M}(v)$ and $\mathcal{M}(b)$) if the atomic write $\mathcal{A}_Q$ succeeds; or otherwise (ii) handle the request as a new one and return newly-assigned $\mathcal{M}(v)$ and $\mathcal{M}(b)$. In the second case P will send data ($\mathcal{D}$) to the newly-assigned volume no matter whether $\mathcal{D}$ has been written to another volume, because data servers are completely agnostic to the status of their blocks and thus have no orphan problems.

Similar to the ordered processing in Fig. 1, the parallel processing in Fig. 2 can also leverage object immutability to avoid overwrite-caused inconsistency. Without the promise of immutability, e.g., if both the proxy server P and the data server S crash at time $t_Q < t < t_S$, then an existing object with the same name as the ongoing one will lose its meta-data because $\mathcal{A}_Q$ has been persisted on Q but $\mathcal{D}_S$ is not yet on S. Immutability can be easily ensured by the clients (like the photo uploader [26]) using unique object names like URLs.

The maintenance of METAX in the Cheetah object store is more complex than in the simplified scenario (Fig. 2), in that Cheetah must provide not only high performance but also high scalability, reliability, and availability. We will introduce the design of Cheetah in §4 and §5, and discuss how Cheetah maintains (and queries) METAX for crash recovery in §6.

4 Hybrid Mapping Architecture

4.1 Overview

Based on the METAX structure we design Cheetah, an efficient directory-based object store that ensures crash consistency without distributed ordering for high I/O performance. The Cheetah object store consists of a manager cluster, a proxy cluster, a data server cluster, and a meta server cluster.

Manager cluster contains an odd number of manager server processes jointly running Paxos [45, 46] as one system manager, which maintains a consistent topology map of the entire system and periodically updates the topology as well as other global information to other servers.

Proxy cluster provides Cheetah clients with the interface of object `put`/`get`/`delete`. Objects are accessed through their globally-unique names.

Data server cluster consists of data storage machines that install disks (divided into physical volumes) for data storage running data server processes to perform raw block I/O.

Meta server cluster consists of meta storage machines that install disks for meta-data storage running meta server processes to maintain and query the METAX structures.

Cheetah adopts n-way replication (storing n replicas) for data and meta-data durability. Note that the replicas for data and for meta-data are organized in different ways.

For data storage, Cheetah organizes the storage capacity by dividing a disk into *physical* volumes. For example, a one-TB disk can be divided into ten 100-GB physical volumes. Cheetah further groups physical volumes into *logical* volumes [26]. A logical volume consists of n physical volumes (from n failure domains) which store exactly the same data.

For meta-data storage, Cheetah applies CRUSH (§2.1) to (i) organize objects into placement groups (PGs) and (ii) map each PG to one primary and $n-1$ backup meta servers. Clearly, it is common for two PGs with the same primary meta server to have different backup meta servers, and vice versa. Primary meta servers are responsible to allocate not only logical volumes but also in-volume space for object data.

4.2 CRUSH-Based Hybrid Mapping

A potential challenge of placing all meta information (meta-data/-log) in METAX is the increasing storage load of the meta service. Consequently, in Cheetah not only the data server cluster but also the meta server cluster needs expansion for the rapidly-growing storage demand. It would be inefficient to rely on a “meta-data directory”, which introduces an extra level of indirection, to maintain the mapping from objects to meta servers. Therefore, Cheetah adopts CRUSH [67] to directly calculate the target meta servers for given objects.

Counterintuitively, it is nontrivial for CRUSH-based meta server calculation to retain migration-free expansion of *data servers*. When adding/removing meta servers, PGs (with their objects’ meta-data) are remapped onto the calculated meta servers. This requires each PG to have its *own group* of logical volumes which are exclusively managed by the PG’s primary meta server. Otherwise, suppose that two PGs (managed by one primary meta server at first) both have objects on one logical volume. If one of the two PGs is mapped onto a new meta server but the other keeps unchanged (which is common for CRUSH), then the logical volume will be managed by different primary meta servers, making consistency guarantees become complex and inefficient.

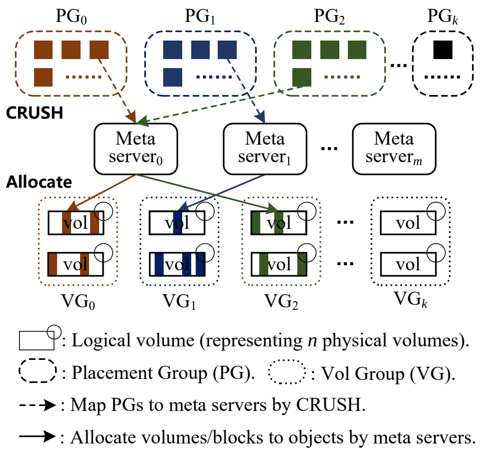


Figure 3: A PG is "CRUSHed" onto a (primary) meta server, which assigns each object in the PG with a logical volume (from the PG's corresponding VG) and in-volume blocks.

To address this problem, we organize the logical volumes into *volume groups* (VGs), each of which belongs 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. When an object belonging to a PG is `put` to Cheetah, the responsible primary meta server first selects a logical volume from the PG's corresponding VG, and then allocates blocks in the logical volume for the object.

The design 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 the meta servers to assign new objects to the new volumes and keeping existing objects unaffected; 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 logical volumes in VGs has changed.

5 Object Storage

This section introduces the basic interface of Cheetah including object `put`/`get`/`delete`. Besides, Cheetah also provides object `list` and namespace `create`/`delete` (not covered here due to lack of space). Note that although immutability disallows overwrites, an object can be modified by deleting it and then putting a new one with the same name [26].

Object put. When putting an object to Cheetah, the client provides the name (e.g., a URL [10]) and data to the proxy. Cheetah processes `put` requests (with n-way replication) by extending the basic procedure of METAX-enabled parallel writes of meta-data and data (§3.2), as shown in Fig. 4.

(1) The proxy server (P) calculates three meta servers by

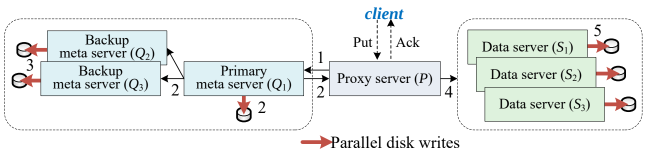


Figure 4: Object `put` of Cheetah (with 3-way replication).

CRUSH (as introduced in §4.2) for the object's PG, among which one is primary (Q_1) and the other two are backup (Q_2, Q_3). P also computes the data checksum and generates a unique request ID (*reqid*). Then P sends the object name, size, checksum, and *reqid* to the primary meta server (Q_1).

(2) The primary meta server (Q_1) takes the following two substeps after receiving the request.

(i) Q_1 selects a logical volume (ID = *lvid*) from the PG's corresponding VG (i.e., volume meta-data $\mathcal{M}(v)$ = *lvid*). It then allocates raw blocks on the selected volume for storing object data using a space allocator (e.g., Ceph's BitMapAllocator [13]). Block-based space allocation effectively reduces *compaction* [26] suffered by large-file-based allocation when reclaiming space of deleted objects, as discussed later in this section. The allocation result is recorded as a list of extents (i.e., block meta data $\mathcal{M}(b)$ = *extents*) of the allocated blocks.

(ii) As discussed in §3.1, the METAX structure ($\mathcal{A}$) consists of $\mathcal{M}(v)$, $\mathcal{M}(b)$, and meta-log ($\mathcal{L}$) that logs the name, proxy, and checksum. Since meta-data is replicated at the granularity of PGs (§4.2), we also log the PG information in $\mathcal{L}$ for meta server crash recovery, as discussed in detail in §6.2. Q_1 performs the following in parallel: (a) returning $\mathcal{M}(v)$ and $\mathcal{M}(b)$ to P, (b) sending $\mathcal{A}$ to the two backup meta servers (Q_2, Q_3), and (c) locally persisting $\mathcal{A}$.

(3) Q_2 and Q_3 write the received $\mathcal{A}$ to their local storage and then acknowledge to Q_1, which will acknowledge to P after persisting $\mathcal{A}$ and receiving all acks.

(4) P looks up the logical volume (with ID = *lvid*) in its local topology map, and sends write request (object data, $\mathcal{M}(v)$, and $\mathcal{M}(b)$) to the corresponding data servers ($S_1 \sim S_3$) of the three physical volumes of the logical volume.

(5) $S_1 \sim S_3$ persist the received data to the blocks (specified by *extents*) on the physical volumes of the logical volume (ID = *lvid*), and then acknowledges to P.

After receiving acks from both the primary meta server and the data servers, P acknowledges to the client and the `put` is *committed*. For efficiency, P does not immediately notify Q_1 to clean the meta-log but instead periodically notifies the meta server about committed `put` for log clean in a batch.

There is no distributed ordering constraint for the multiple writes in Steps (2)ii(c), (3), and (5) performed in parallel on the n meta servers and n data servers. We will discuss how Cheetah ensures consistency of immutable objects by enforcing local write atomicity on meta servers in §6.

Object get. When getting an object, the client provides the object name to the proxy. The processing is as follows.

(1) The proxy server (P) calculates the primary meta server (Q_1) for the name by CRUSH, and sends a query to Q_1.

(2) Q_1 searches locally for the name and retrieves the corresponding meta-data ($\mathcal{M}(v)$ = *lvid*, $\mathcal{M}(b)$ = *extents*, and checksum c). Q_1 then returns *lvid*, *extents* and c to P.

(3) P looks up the logical volume (ID = *lvid*) in its local topology map, and then sends a read request with the meta-data (*lvid*, *extents*) to the data server of one of the n physical volumes of the logical volume.

(4) The data server reads the blocks (specified by *extents*) from the corresponding physical volume, and returns to P.

(5) The proxy server P verifies the checksum of the data and (if correct) returns the data to the client.

Similar to Haystack, Cheetah can optimize the `get` performance if the client is reading an object previously put by itself. When processing a `put` request, the proxy server could acknowledge to the client with the meta-data (*lvid* and *extents*), which will be cached by the client and taken in subsequent `get` requests. After receiving such `get` requests, the proxy server will perform the above steps (2) and (3)(4) in parallel, and use the checksum (c) from Q_1 to prevent malicious clients from tampering *lvid* and *extents*.

Object delete. Existing file-based volume space allocation approaches either append objects to large files (e.g., in Haystack [26]), or allocate each object with a small file (e.g., in Swift [20] and MinIO [18]). For processing a `delete` request, the large-file approach is lightweight in that only a flag needs to be set in the meta-data. However, this approach requires *compaction* [26] where the space of deleted objects must be reclaimed by moving the remaining objects to new files, which causes performance degradation of in-compaction I/O. In contrast, the small-file approach needs no compaction for reclaiming, but it is costly in processing object I/O requests due to nontrivial per-object filesystem overhead.

Owing to MetaX, the data servers in Cheetah are object-agnostic and can provide ultralight block service where raw blocks can be directly read from and written to volumes without intermediate file abstraction. This allows Cheetah to (i) efficiently process a `delete` request by simply updating the meta-data on meta servers and (ii) effectively reduce compaction. The processing of a `delete` is as follows.

(1) The proxy server (P) gets the primary meta server (Q_1) for the object name via CRUSH, and sends `delete` to Q_1.

(2) Q_1 looks up *lvid* and *extents* in its local storage for the name. Q_1 then deletes the corresponding meta-data/-log from its local storage, and updates the bitmap (used by the space allocator) of the logical volume with ID = *lvid* by clearing the bits specified by *extents*. Meanwhile, Q_1 sends the `delete` request to the $n-1$ backup meta servers of the object, which will process the request in the same way.

(3) After completing its local processing and receiving acks from all backup meta servers, Q_1 acknowledges to the proxy server, which finally acknowledges to the client.

Block-based allocation allows immediate reuse of the reclaimed space once the corresponding bits of the deleted object data are updated in the bitmap. Since the relatively small objects in OLDI applications tend to have similar sizes, we expect Cheetah to find appropriate reclaimed blocks (probably with marginal fragmentation) for newly-written objects, instead of allocating new blocks. Like Haystack, Cheetah can resort to compaction if the overall fragmentation is high, which is rare for deletion-intensive workloads of similar-sized objects. In contrast, appending objects to files precludes such optimization due to the intermediate file abstraction. Cheetah is particularly suitable for deletion-intensive workloads (§7) where delete operations are common but unpredictable. Note that TTL (time-to-live) based expiration [8] (for avoiding fragmentation) is not suitable for this scenario, because it requires knowledge about deletion times ahead of time.

6 Crash Recovery

Cheetah follows Vertical Paxos [46] and its variations [55, 68] to guarantee consistency and durability under various server crashes (possibly with false positives due to, e.g., network problems) and power loss. A manager cluster (with $2f+1$ servers jointly running Paxos as one system manager) is responsible for managing the global topology map, so that n-way replication (storing meta-data/data on n meta/data servers) can tolerate $f = n-1$ failures and ensure availability (with partial synchrony assumption due to FLP impossibility [36]). Specifically, Cheetah can recover from (i) f simultaneous crashes of the physical volumes of a logical volume and (ii) f simultaneous crashes of the meta servers of a PG. The primary meta servers are responsible for coordinating crash recovery, including not only recovery of lost object data/meta-data but also completion or revocation.

6.1 Topology Map

Table 1 lists the global information maintained in the topology map. In the hybrid mapping architecture (§4.2) of Cheetah, objects are first mapped onto the meta servers by CRUSH and then mapped onto the volumes of data servers by the meta servers, which respectively require (i) the topology information of proxy/data/meta servers and (ii) the membership of logical volumes in VGs as well as the logical-to-physical volumes mapping. Besides, the cluster manager also maintains the view number [57] and lease [39], as introduced below.

All servers must maintain the topology map in a consistent way. Otherwise, an outdated server may send stale messages which will cause inconsistency. To address this problem, the cluster manager maintains a *view number* which is incremented every time the topology map changes. The view number is disseminated to all servers and piggybacked with every request, so as to reach a consensus about the current topology map. A request can be processed by a server only if the server has a matching view number for the request. The lagged-behind server (with a lower view number) will update its view number and topology map coordinated by the cluster

Table 1: Global information in the topology map.

Server topology	$P_1, P_2, \cdots, Q_1, Q_2, \cdots, S_1, S_2, \cdots$
Volume info of VGs	VG_1: {vol}, VG_2: {vol}, $\cdots$
View number	i
Lease	$[t_1, t_2), [t_2, t_3), \ldots$

manager. Note that data servers need no topology information except the view number.

A get request can be served by the primary meta server and any one of the n corresponding data servers. In order to support single-server read while ensuring consistency, Cheetah also leverages a lease mechanism [27, 39, 53] for topology map maintenance. A lease is a period of time during which the map will not change. The cluster manager renews and disseminates the lease periodically (e.g., for every 10 seconds), and a meta server will not answer requests if its local lease has expired. Without lease, it is possible that a proxy server P has a (temporarily) stale view number and accepts the meta-data of a get request from an outdated meta server Q. Then inconsistency occurs when the topology (without Q) has been updated on all servers except P and the requested object has been deleted/modified on its new meta servers.

6.2 MetaX KV

Theoretically the MetaX structure can be stored in any local storage on meta servers that supports atomical write of the meta-data/-log. However, Cheetah has additional *query* requirements for high efficiency, and thus we adopt a KV store as the local storage for meta servers.

First, the meta servers need to quickly retrieve the meta-data for given keys to support efficient get/delete, which requires Cheetah to store the meta-data ($\mathcal{M}(v)$ and $\mathcal{M}(b)$) in a KV. Second, if a proxy server P crashes, the meta servers must find all the ongoing put requests of P to retrieve the relevant meta-data, which requires Cheetah to store the meta-log for P in a KV. Third, if a meta server Q crashes, for each PG on Q the meta-data of each object in the PG must be restored by the PG's remaining meta servers, which requires Cheetah to maintain the PG information in a KV. We integratively design the keys/values of the KVs, as shown in Table 2.

The first KV is designed to store the meta-data, with key = OBMETA_*objectname* and value = {*attrbutes, lvid, extents, checksum*}. In the key, OBMETA is a prefix for "object meta-data" (for prefix matching queries), and *objectname* 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 ($\mathcal{M}(b) = extents$), the object data's *checksum* (received from the proxy server), and the optional *attributes*.

The second KV logs the information of the requesting proxy server (P), with key = PXLOG_*pxid_reqid* and value = {*objectname, oplogkey*}. In the key, PXLOG stands for "proxy log", *pxid* is the proxy server ID, and *reqid* is the request ID generated by the proxy server. The value includes the unique name of the object (*objectname*), and the key of the PG log

Table 2: MetaX KVs.

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

(*pglogkey* = PGLOG_*pgid_opseq*) introduced below.

The third KV logs the PG information of the put request, with key = PGLOG_*pgid_opseq* and value = {*objectname, reqlogkey*}. In the key, PGLOG stands for "PG log", *pgid* is the PG's ID (by hashing *objectname*), and *opseq* is a sequence number (generated by the meta server) which is monotonically increased for every put request in the PG. The value includes the name of the object (*objectname*), and the key of the proxy log (*pxlogkey* = PXLOG_*pxid_reqid*) introduced above.

The three MetaX KVs must be atomically written for each put, which is well supported by modern KV stores like RocksDB [19] and Badger [7]. Besides, Cheetah maintains an in-memory bitmap (where one bit represents the status of a block) for each logical volume to facilitate the selection of volumes and allocation of blocks. Cheetah periodically flush the bitmaps to disks each time when the corresponding PG logs are cleaned in a batch. The in-memory bitmap is consistent with its on-disk version plus the accumulated updates (i.e., *extents* in the first KV) since it is last flushed.

6.3 Handling Various Crashes

Cheetah uses heartbeats [43] to detect server failures. If the cluster manager cannot receive heartbeats from a server in a timely manner, it will remove that server, update the topology map, and disseminate the new map after the current lease expires. Servers that are removed from the new map or hold stale view numbers/expired leases cannot process requests.

Meta server crash. If a meta server Q becomes unavailable, Cheetah will wait for a short period of time t expecting Q to recover (while marking the affected PGs as *readonly*). If Q cannot recover in t, then the manager will consider Q crashed and update the topology map with increased view number. For each affected PG, CRUSH will calculate a new meta server which replaces Q by restoring the KVs, and select a new primary for its VG.

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

Proxy server crash. If a proxy server P (ID = *pxid*) crashes, each primary meta server Q that is processing `put` from P will look up the proxy logs in its KVs with prefix key PXLOG_*pxid*. Each retrieved proxy log records OBMETA_*objectname* in the value. Q further looks up the meta-data in the KVs with key OBMETA_*objectname*, and retrieves *lvid*, *extents* and *checksum* from the value. Then Q will send *lvid* and *extents* to the data servers which return the checksums of the blocks.

Q compares the returned checksums with the retrieved *checksum*. If at least one checksum (say, from data server S) is correct, then the object data has been persisted on S. Q will complete the processing of the `put` by sending the KVs to the backup meta servers and asking S to recover the data. Q will also clean the current proxy log and the relevant PG log with key = *pglogkey* (stored in the value of the proxy log). Otherwise, if none of the checksums is correct, then Q has to revoke the incomplete operations by removing the KVs.

Data server crash. If a storage data server S (with its corresponding physical disk and the physical volumes on that disk) 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 logical volumes when allocating volumes for new `put`.

If S recovers in a short period, then the logical volumes will be marked as writable again and S will resume its service normally. Otherwise, the manager will change the topology map and assign a new data server S' (with its new physical disk and the physical volumes on that disk) to replace S, and notifies the primary meta servers that are responsible for the affected logical volumes. The lost data will be restored to each new physical volume from one of its $n-1$ healthy physical volumes. Finally S' resumes the normal block service.

A special case is that the data server S crashes when it is writing data of an ongoing `put` request. In this case, the proxy server will re-issue the `put` (marked as RE-DATA) with the new view number. The responsible meta server will atomically (i) select a new logical volume from the PG's corresponding VG to allocate blocks, and (ii) revoke the previous allocation on the problematic volume by resetting the bits in the bitmap.

Power loss. If all servers are down due to a power loss, after reboot the meta servers will first negotiate with each other for the PG logs to synchronize the METAX KVs by comparing the *opseq* in the key, and then compare *checksum* in the meta-data KV 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 the meta servers will revoke the `put` by deleting the corresponding KVs. After all comparisons and revocations are done, the proxy servers can resume the normal object I/O service.

6.4 Correctness

This subsection proves that Cheetah ensures crash consistency: if a `put` request is committed then future `get` requests will see the (most recent) data, unless a `delete` is committed thereafter. Different from existing work, Cheetah writes meta-data 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.

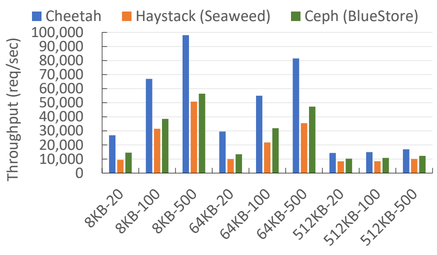
If a `put` request is committed, then its meta-data and data should be replicated respectively by n meta servers and n data servers. Without topology map update, a `get` request will obtain the matching meta-data and object data from the primary meta server and one corresponding data server.

Now we argue that Cheetah ensures crash consistency after updating the topology map, even if Cheetah writes meta-data and data in parallel. Suppose that an ongoing `put` (req_p) is issued in view i, and that $f = n-1$ meta severs storing the meta-data of req_p and f data servers storing the object data of req_p either crash permanently or are partitioned due to network problems. The manager cannot collect their heartbeats and thus initiates the topology update procedure. It (i) updates the topology map by removing the f problematic meta servers and replacing the f problematic data servers with f data servers of healthy physical volumes within the same VG, (ii) disseminates the new map with view number $i+1$, and (iii) asks the new primary meta server Q' to coordinate the recovery. Q' will contact a remaining meta server Q (which might be itself) and a remaining data server S in view i.

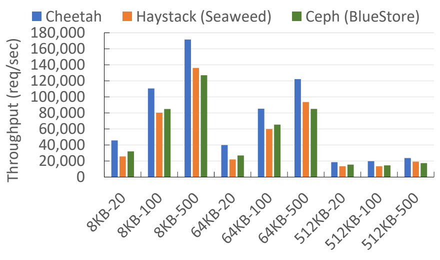
We discuss two cases. First, if req_p is already committed by the proxy server in view i, then both Q and S should have written the meta-data and object data of req_p in view i. Because of object immutability, the meta-data of req_p will not be replaced by a new `put` at Q, and thus Q and S will transfer the meta-data/data of req_p to the new servers in view $i+1$. The object can be accessed after recovery. Note that `delete` request does not introduce distributed write operations owning to the METAX structure. Either a `delete` succeeds in view i after req_p and Q should not store meta-data of req_p anymore; or req_p can be recovered in view $i+1$ with the help of Q.

Second, if the meta-data of req_p is not written on Q or the object data of req_p is not written on S, then the proxy server had not committed 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 meta-data of req_p is written on Q, but S does not have the data, then Q will abort req_p and revoke its processing since the checksum does not match. Also note that in this case the proxy server cannot commit req_p in view i anymore, because either Q or S will reject req_p due to the mismatching of view numbers (req_p is in view i, but Q or S 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.

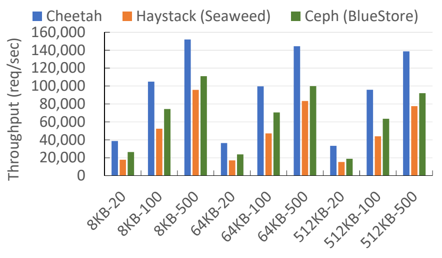
Finally, suppose that a `get` request (req_g) for the same object is issued after the topology map update. If req_g is with view i then it will not be served by any servers because of the lease mechanism. So req_g must be re-issued in view $i+1$ and thus the corresponding meta/data servers in view $i+1$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



(a) PUT Throughput

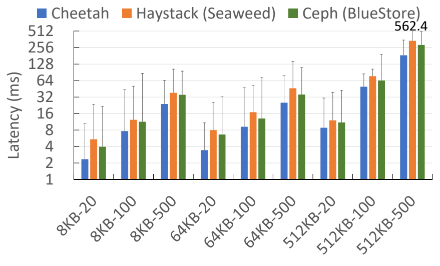


(b) GET Throughput

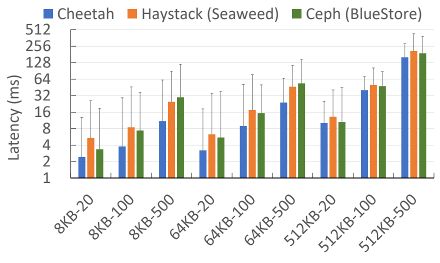


(c) DELETE Throughput

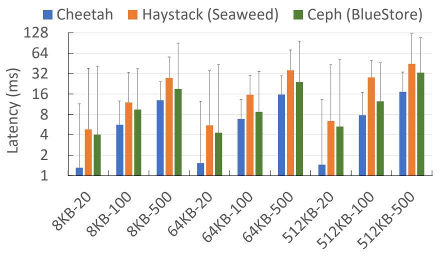
Figure 5: Throughput of Cheetah, Haystack and Ceph. For example, 8KB-20 represents data size of 8KB and concurrency of 20.



(a) PUT Latency



(b) GET Latency



(c) DELETE Latency

Figure 6: Mean and 95th percentile Latency of Cheetah, Haystack and Ceph.

7 Evaluation

We evaluate Cheetah through micro benchmarks and trace-driven experiments. Our testbed consists of sixteen machines which are connected to a two-level FatTree Ethernet with no oversubscription [52] (emulated by two Mellanox SN2700 100GbE switches [25]). Each machine has two dual 8-core Xeon E5-2650 2.30GHz CPUs and 256GB RAM. The operating system is CentOS 7.8. Three machines run the clients and three run the proxy servers. One machine runs the cluster manager. We have six data machines and three meta machines, and adopt three-way replication for both object data and meta-data. A data machine has four Intel 750 PCIe 1.2TB SSDs for object data storage each being divided into 120 10GB physical volumes. Totally we have $120 \times 4 \times 6 = 2880$ physical volumes and $2880/3 = 960$ logical volumes. We organize the logical volumes into 240 volume groups (VGs), each having four logical volumes and corresponding to one PG. A meta machine has two Intel 750 PCIe 400GB SSDs for meta-data storage. We use RocksDB (v6.7) [19] for METAX KVs and modify the Ceph BitMapAllocator [13] for allocation of volume blocks. RocksDB adopts the default configurations [16]. The block size (`min_alloc`) of the BitMapAllocator is (by default) 8KB, and thus the size of a bitmap is $10\text{GB}/8\text{KB}/8 \approx 160\text{KB}$.

We compare Cheetah with Haystack [26], the state-of-the-art *directory*-based object store. Since Haystack is closed-source, we evaluate Haystack's open-source implementation SeaweedFS [15] (v1.7) instead. The configuration of Haystack is the same as that of Cheetah, except that Haystack has three directory servers instead of meta servers and does not support CRUSH-based directory discovery (§4.2). For fairness we statically divide the object name space into three subspaces each being mapped to one directory server, so as to avoid extra discovery overhead. We also evaluate Ceph RGW (v15.2) [14, 66], the state-of-the-art *hash*-based object store. We use BlueStore [6] as the backend, which stores data on raw disks and adopts CRUSH to directly calculate the target volumes without accessing directory/meta servers. We do not compare Cheetah to MapX [64] (which realizes time-dimension mapping for CRUSH on Ceph-RBD [4] to avoid migration), because (i) MapX cannot support common object storage due to the overwhelming per-object timestamp overhead and (ii) MapX performs no better than Ceph.

7.1 Micro Benchmarks

We compare the throughput (number of requests per second) and latency (request completion times) of Cheetah, Haystack and Ceph. The clients run Intel COSBench [17] to `put` 10 million objects with various data sizes and concurrency (number of ongoing requests). §7.3 will test more concurrency.

The results are shown in Figs. 5a and 6a, where Cheetah outperforms Haystack by 67.5% $\sim$ 1.96$\times$ and 36.7% $\sim$ 1.32$\times$ in the throughput and latency, respectively. The advantage is mainly because Cheetah avoids the ordering constraint of meta-data/data writes compared to Haystack. Besides, Cheetah also benefits from its raw block service which avoids filesystem overhead on data servers. Figs. 5a and 6a also show that Cheetah even outperforms the hash-based Ceph which should theoretically have better performance, considering Ceph saves the cost of accessing meta servers. Ceph's inefficiency is mainly because of the complex layered design for its rich features affecting concurrency [48]. Ceph

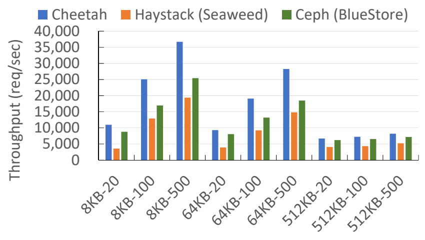


Figure 7: Read-modify-write.

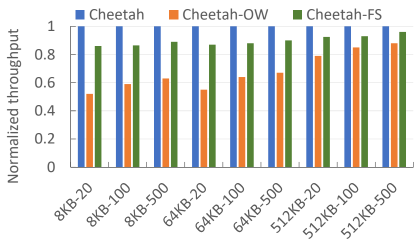


Figure 8: Impacts of ordering and FS.

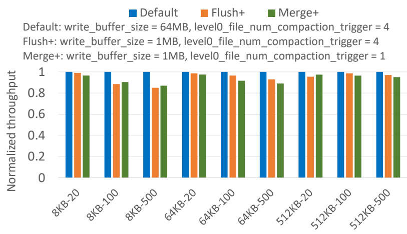


Figure 9: RocksDB configurations.

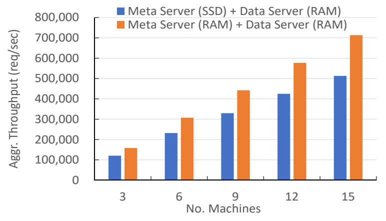


Figure 10: Meta service scalability.

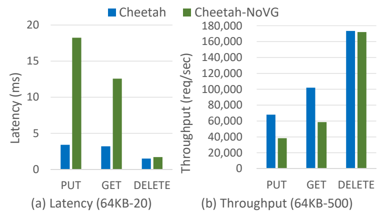


Figure 11: Meta service expansion.

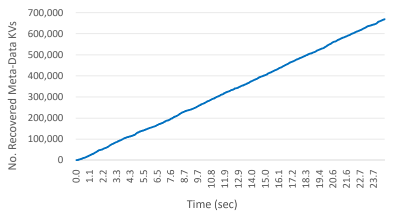


Figure 12: Meta server recovery.

has to either write logs for small ($\leq$32KB) objects or enforce *local ordering* constraints for large objects on its data servers, so as to support its general-purpose upper-layer applications (Ceph-RBD [4] and Ceph-FS [5]) which allow overwrites.

We measure the `get` performance. The clients first `put` 10 million objects and then `get` 100,000 objects (chosen uniformly at random). In this experiment the cache on data servers takes little effect because of the relatively low ratio (1%) of objects read by the clients. Client-side optimization (§5) is disabled to avoid uncertainty. The results are shown in Figs. 5b and 6b, where Cheetah outperforms Haystack by 23.5% $\sim$ 81.7% and 23.9% $\sim$ 1.25$\times$ in the throughput and latency, respectively. The advantage is mainly because Cheetah only needs to read raw data blocks while Haystack has to read both meta-data and data in its filesystem on data servers. Cheetah also outperforms Ceph because reading meta-data on Ceph data servers is complex and expensive.

We measure the `delete` performance. Similar to the test for `get`, the clients first `put` 10 million objects and then `delete` 100,000 randomly-chosen objects. The results are shown in Figs. 5c and 6c. Cheetah outperforms Haystack and Ceph in both throughput and latency, because Cheetah only needs to delete the meta-data from RocksDB on its meta servers.

Immutable objects can be modified by first deleting it and then writing the modified one [26]. We write a custom benchmark tool to compare the read-modify-write performance of Cheetah, Haystack, and Ceph, using the same configurations in the previous `put` test. We first `put` 10 million objects, and then randomly read-modify-write 100,000 of them. Fig. 7 shows the result where Cheetah always achieves the highest throughput. For Cheetah and Haystack, the clients read an object, delete it, and write back the new one, and thus their performance is jointly affected by `put`/`get`/`delete`. Ceph has no restriction on object immutability and thus it can perform read-modify-write by overwriting objects without `delete`.

7.2 Impacts of Design Components

MetaX enables parallel meta-data/data writes without ordering. We evaluate the impact of parallel writes by comparing Cheetah to Cheetah-OW (its variation with *ordered-writes*), where the proxy servers send the volume/block meta-data and data to the data servers after receiving acks from the meta servers, instead of right after receiving the meta-data. Fig. 8 shows the normalized throughput where Cheetah outperforms Cheetah-OW by up to 92.3%, demonstrating the effectiveness of the removal of the ordering constraint. The advantage of parallel writes is relatively low when object size = 512KB, because the data servers take more time to write object data which alleviates the impact of distributed ordering.

MetaX facilitates the design of the raw block service on data servers. To quantify the performance contribution 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 the ext4 filesystem and use a large file on each volume to write object data. Since the test is only for comparison between filesystem and raw block I/O in processing `put`, Cheetah-FS could simply ignore the received block meta-data ($\mathcal{M}(b)$). Fig. 8 also shows the normalized throughput of Cheetah-FS. The advantage of Cheetah over Cheetah-FS is smaller than that over Cheetah-OW, because in this `put`-only experiment the data servers of Cheetah-FS can write data to large files without significant overhead.

Cheetah uses RocksDB for MetaX KV storage. RocksDB writes the KVs both in its in-memory Memtable (of size `write_buffer_size`) and in a write ahead log (WAL). When a Memtable is full, it is *flushed* to a Sorted Sequence Table (SST) file and the log is cleaned. SST files are organized in a sequence of levels starting from Level-0. When one level reaches its trigger, SST files will be merged to the next level (called *layer merge*). The file number trigger for Level-0 is controlled by `level0_file_num_compaction_trigger`.

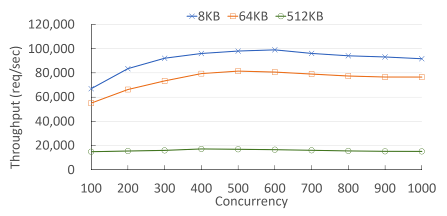


Figure 13: TP. with various concurrency.

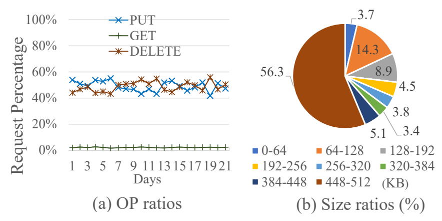


Figure 14: Three-week trace.

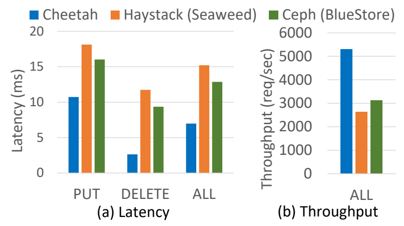


Figure 15: Trace-based comparison.

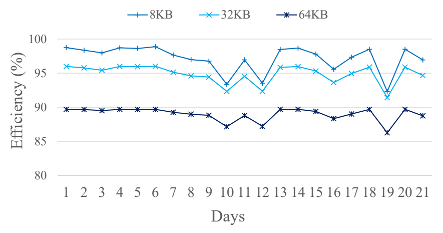


Figure 16: Storage Efficiency.

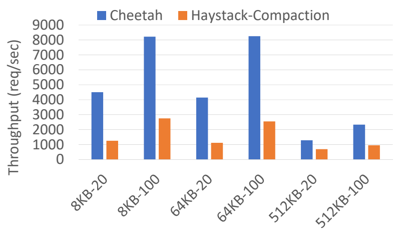


Figure 17: In-compact Haystack.

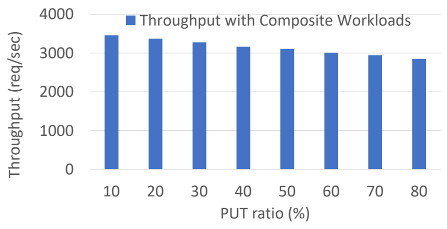


Figure 18: Composite workloads.

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. 9 shows the normalized throughput compared to the original configuration when we `put` 10 million objects. The result shows that increasing flush and merge rates only has small impact on the overall performance.

7.3 Scalability & Expandability

The CRUSH-based meta service of Cheetah is vital for handling the increasing load of MetaX storage. To evaluate the scalability of the meta service, we re-deploy all the 30 (24 1.2TB + six 400GB) SSDs to the 15 machines (except the one for the cluster manager), each having two SSDs for meta-data storage. Each of the 15 machines also runs a client, a proxy server, as well as a virtual data server using a RAM disk [9] (instead of SSDs) for object data storage. We use $m = 3, 6, 9, 12, 15$ machines to form the storage cluster, respectively, and all the m clients jointly `put` 10 million 8KB objects. We measure the aggregate throughput of the m-machine cluster. The result is shown in Fig. 10, where the throughput of Cheetah increases linearly with the number of machines. For comparison, in Fig. 10 we also evaluate the theoretical upper bound for the throughput of Cheetah (both meta servers and data servers using RAM disks), showing that Cheetah's meta service achieves near-optimal scalability.

Cheetah leverages volume groups (VG) for migration-free expansion. We evaluate the effectiveness of VG with the same configuration as that in §7.1, except that we use five data machines and reserve the sixth machine 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, which uses two SSDs for meta-data storage (the same as the existing three meta machines). CRUSH remaps 1/4 the PGs to the new meta machine, after which we measure the performance of `put`/`get`/`delete` for 64KB objects of Cheetah (with VG) and Cheetah-NoVG (without VG) on the unaffected PGs. The results are depicted in Fig. 11, where Cheetah significantly outperforms Cheetah-NoVG for `put` and `get`. This is because in Cheetah-NoVG 1/4 the logical volumes are reassigned to the new meta machine, which inevitably causes huge data migration (§4.2). In contrast, logical volumes in Cheetah belong to VGs (each corresponding to a PG), which avoids PG-remapping-caused migration. Cheetah and Cheetah-NoVG have similar performance for `delete`, since `delete` is performed only on meta servers. The `delete` throughput is slightly higher than that in Fig. 5c owing to the new meta machine.

We test the CRUSH-based meta server recovery. We (i) write 8KB objects with concurrency = 100 for 10 seconds, (ii) disconnect a meta machine, and (iii) re-connect it as a new meta machine. After waiting for a short period of time, the manager will consider the old meta machine crashed and disseminate the new topology map in the next lease. Then the meta-data of the affected PGs will be recovered to the new meta servers' disks. The recovery procedure is shown in Fig. 12, where all meta-data is recovered within 24 seconds, proving the effectiveness of Cheetah's recovery mechanism.

We measure the impact of concurrency on the throughput of Cheetah by increasing client-side concurrency from 100 to 1000, for various object data sizes. The result is shown in Fig. 13, where the throughput increases as concurrency increases from 100 to (around) 500. Afterwards the throughput slightly decreases while the mean latency greatly increases up to a few seconds (not shown here). The impact of concurrency (> 100) is more prominent for small sizes (8KB/64KB) than for large sizes (512KB), because large objects can (almost) saturate Cheetah with relatively-low concurrency.

7.4 Trace Driven Evaluation

We compare Cheetah with Haystack and Ceph through trace-driven experiments. We collect a three-week trace of the workload of a subset of a production cluster, which stores objects for various OLDI applications of one of the largest ride-hailing services in the world, such as audio/video recording and analysis, face identification, sentiment analysis, status assessment of drivers/passengers (where privacy-safety dilemma is out of the scope of this paper). These applications store their data as small ($\leq$ 512KB) objects. To accommodate all objects in the trace, in this subsection we install six 6TB HDDs (instead of four 1.2TB SSDs) on each data machine and two 1TB HDDs (instead of two 400GB SSDs) on each meta machine. All other configurations are the same as before.

Fig. 14a 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 life cycle (ranging from a few hours to months). The ratios of different object sizes are shown in Fig. 14b. Different from read-intensive applications (like social media [26, 55]), a wide range of recently-emerged OLDI systems are *write-intensive* [22, 69] in that a large amount of objects are continuously written but only a few will be selectively read for computation by multiple registered analysis applications. For instance, writes are predominant in the video analysis application in the trace, where only a small fraction of all recorded frames are sampled and analyzed [56].

We use the custom benchmark tool (ignoring the timestamps in the trace) to replay the recorded I/O requests (with concurrency $= 20$). For a `delete`, if the object is not `put` in the trace we first `put` it before the test. We evaluate the latency and throughput for Cheetah, Haystack, and Ceph. The results (Fig. 15) show that Cheetah substantially outperforms others in real production workloads, even though the sizes of a large fraction (68.6%) of objects $>$ 256KB.

For data servers, Cheetah's raw block storage is more preferable than Haystack's filesystem for production workloads because Cheetah can directly reuse the reclaimed space of deleted objects (with marginal fragmentation), which is impossible for Haystack without compaction. Cheetah will also resort to compaction if the overall fragmentation severely affects the storage efficiency (total object size/occupied space). We evaluate the storage efficiency of Cheetah at the end of each hour during the replay of the three-week trace. Fig. 16 shows the maximum efficiency for each day's 24 hours. In addition to the default block size of 8KB, we also replay the trace using block sizes of 32KB and 64KB. Cheetah achieves high storage efficiency ($>$ 85%) even for 64KB block size, mainly because the written/deleted objects have similar sizes.

In contrast to Cheetah, Haystack has to perform compaction to reclaim space. To understand the negative impact of Haystack's compaction, we evaluate its in-compaction performance. We first fill the system with objects and randomly delete objects until compaction occurs (observed via `iostat`) in Haystack. We then evaluate the in-compaction `put`. The result is shown in Fig. 17, which also includes the performance of (compaction-free) Cheetah for comparison. The advantage of Cheetah gets much higher when Haystack is in compaction, which is inevitable (due to the intermediate file abstraction) for workloads where `delete` operations are common for objects with unpredictable lifecycles.

We use YCSB benchmark tool [12] to generate workloads with combined I/O requests. The `delete` ratio is 10%, the `put` ratio varies from 10% to 80%, and accordingly the `get` ratio varies from 80% to 10%. Object sizes are chosen (uniformly at random) between 4 $\sim$ 512 KB. The result is shown in Fig. 18. The throughput slightly decreases as the `put` ratio increases, proving Cheetah adapts to a wide range of workloads.

8 Related Work

Object stores. In addition to Facebook's Haystack (§2.3), Twitter uses virtual buckets to store its photos [3], LinkedIn's Ambry [55] adopts logical grouping and asynchronous replication to realize a geo-distributed object store [61], and Facebook's F4 [54] uses erasure coding [49] to reduce replication factor of warm data. Object storage also becomes a public service for OLDI applications, such as Amazon S3 [2], Microsoft Azure [28], and Ali OSS [21]. S3 stores each object as a small file and provides *eventual consistency* [34]. These systems are only optimized for read-intensive workloads [1].

Distributed KV stores and filesystem. Key-value storage [7, 19, 30, 35, 44] provides an interface for accessing the values associated with keys. Since KV needs to support various queries such as prefix key matching and range query, they usually require relatively complex structures for indexing and optimization [29], and thus have problems of amplification and inefficiency for object storage. They are usually used as building blocks for object meta-data. Distributed file storage [31, 33, 37, 40–42, 47, 51, 60] needs to support the complex file abstraction and has similar problems for objects with KVs.

Consistency and ordering. NoFS [32] removes the need for any ordering to disk at all for high I/O performance. But NoFS might cause a recovered file system to contain data from partially completed operations. NoFS cannot support local atomic actions (like `rename`). OptFS [31] analyzes the crash consistency caused poor I/O performance in local journaling file systems. The design of METAX is inspired by OptFS, but we are targeted to object storage where (i) the distributed write ordering requires both data and meta-data to be replicated and (ii) the object immutability allows us to ensure consistency by enforcing the (much simpler) local atomicity.

9 Conclusion

This paper presents Cheetah, an efficient object store based on the METAX structure containing all meta-data/-log of an ongoing `put`. Cheetah outperforms existing systems by parallelizing meta-data and data writes with consistency guarantees. We will improve Cheetah with chain replication [63], asynchronous replication [65] for hybrid fault models [50, 58], immutability guarding mechanism, and fast query [59, 70]. The key components of Cheetah are available at [11].

References

[1] https://aws.amazon.com/blogs/storage/protecting-data-with-amazon-s3-object-lock/.

[2] https://aws.amazon.com/s3/.

[3] https://blog.twitter.com/engineering/en_us/a/2012/blobstore-twitter-s-in-house-photo-storage-system.html.

[4] https://ceph.com/ceph-storage/block-storage/.

[5] https://ceph.com/ceph-storage/file-system/.

[6] https://ceph.io/community/new-luminous-bluestore/.

[7] https://dgraph.io/blog/post/badger/.

[8] https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/howitworks-ttl.html.

[9] https://en.wikipedia.org/wiki/RAM_drive.

[10] https://en.wikipedia.org/wiki/URL.

[11] https://github.com/anonymous-nicer/metax.

[12] https://github.com/brianfrankcooper/YCSB/wiki/Implementing-New-Workloads.

[13] https://github.com/ceph/ceph/blob/master/src/os/bluestore/Allocator.cc.

[14] https://github.com/ceph/ceph/tree/master/src/rgw.

[15] https://github.com/chrislusf/seaweedfs.

[16] https://github.com/facebook/rocksdb/wiki/Setup-Options-and-Basic-Tuning.

[17] https://github.com/intel-cloud/cosbench.

[18] https://min.io/.

[19] https://rocksdb.org/.

[20] https://swift.org.

[21] https://www.alibabacloud.com/zh/product/oss.

[22] https://www.alluxio.io/blog/accelerating-write-intensive-data-workloads-on-aws-s3/.

[23] https://www.facebook.com/.

[24] https://www.linkedin.com/.

[25] https://www.mellanox.com/related-docs/prod_eth_switches/PB_SN2700.pdf.

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

[27] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 335–350, USA, 2006. USENIX Association.

[28] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011.

[29] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.

[30] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *Acm Transactions on Computer Systems*, 26(2):1–26, 2008.

[31] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 228–243. ACM, 2013.

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

[33] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146. ACM, 2009.

[34] Sudipto Das. Rethinking eventual consistency: Can we do better? In Bipin V. Mehta, Nikos Mamoulis, Arnab Bhattacharya, and Maya Ramanath, editors, *19th International Conference on Management of Data, COMAD*

2013, Ahmedabad, India, December 19-21, 2013, page 5. Computer Society of India, 2013.

[35] Giuseppe Decandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *Acm Sigops Operating Systems Review*, 41(6):205–220, 2007.

[36] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[37] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP*, pages 29–43, 2003.

[38] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 202–210, New York, NY, USA, 1989. ACM.

[39] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *SIGOPS Oper. Syst. Rev.*, 23(5):202–210, November 1989.

[40] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Analysis of hdfs under hbase: A facebook messages case study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 199–212, 2014.

[41] John H Hartman and John K Ousterhout. The zebra striped network file system. *ACM Transactions on Computer Systems (TOCS)*, 13(3):274–310, 1995.

[42] Dean Hildebrand and Peter Honeyman. Exporting storage systems in a scalable manner with pnfs. In *22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05)*, pages 18–27. IEEE, 2005.

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

[44] Avinash Lakshman and Prashant Malik. Cassandra:a structured storage system on a p2p network. In *Proc Acm Sigmod International Conference on Management of Data*, 2009.

[45] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[46] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. Technical Report MSR-TR-2009-63, May 2009.

[47] Andrew W Leung, Shankar Pasupathy, Garth R Goodson, and Ethan L Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX annual technical conference*, volume 1, pages 2–5, 2008.

[48] Huiba Li, Yiming Zhang, Dongsheng Li, Zhiming Zhang, Shengyun Liu, Peng Huang, Zheng Qin, Kai Chen, and Yongqiang Xiong. Ursa: Hybrid block storage for cloud-scale virtual disks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 15. ACM, 2019.

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

[50] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. Xft: Practical fault tolerance beyond crashes. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 485–500, Berkeley, CA, USA, 2016. USENIX Association.

[51] Marshall K McKusick, William N Joy, Samuel J Leffler, and Robert S Fabry. A fast file system for unix. *ACM Transactions on Computer Systems (TOCS)*, 2(3):181–197, 1984.

[52] James Mickens, Edmund B Nightingale, Jeremy Elson, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, Osama Khan, and Krishna Nareddy. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 257–273, 2014.

[53] Iulian Moraru, David G. Andersen, and Michael Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–13, New York, NY, USA, 2014. Association for Computing Machinery.

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

[55] Shadi A. Noghabi, Sriram Subramanian, Priyesh Narayanan, Sivabalan Narayanan, Gopalakrishna Holla, Mammad Zadeh, Tianwei Li, Indranil Gupta, and Roy H. Campbell. Ambry:linkedin's scalable geo-distributed object store. In *International Conference on Management of Data*, pages 253–265, 2016.

[56] Peter Ochs, Jitendra Malik, and Thomas Brox. Segmentation of moving objects by long term video analysis. *IEEE transactions on pattern analysis and machine intelligence*, 36(6):1187–1200, 2013.

[57] Brian M. Oki and Barbara H. Liskov. 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*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[58] Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. Visigoth fault tolerance. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 8:1–8:14, New York, NY, USA, 2015. ACM.

[59] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 497–514. ACM, 2017.

[60] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 237–248. IEEE Press, 2014.

[61] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. Optimistic causal consistency for geo-replicated key-value stores. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pages 2626–2629. IEEE, 2017.

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

[63] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 91–104. USENIX Association, 2004.

[64] Li Wang, Yiming Zhang, Jiawei Xu, and Guangtao Xue. Mapx: Controlled data migration in the expansion of decentralized object-based storage systems. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020.

[65] Yang Wang, Lorenzo Alvisi, and Mike Dahlin. Gnothi: Separating data and metadata for efficient and available storage replication. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 38–38, Berkeley, CA, USA, 2012. USENIX Association.

[66] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, 2006.

[67] Sage A Weil, Scott A Brandt, Ethan L Miller, and Carlos Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 31–31. IEEE, 2006.

[68] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. Rados: A scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*, PDSW '07, page 35–44, New York, NY, USA, 2007. Association for Computing Machinery.

[69] Lin Wu, Qingfeng Zhuge, Edwin Hsing-Mean Sha, Xianzhang Chen, and Linfeng Cheng. Boss: An efficient data distribution strategy for object storage systems with hybrid devices. *IEEE Access*, 5(2):23979–23993, 2017.

[70] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. Lsm-trie: an lsm-tree-based ultra-large key-value store for small data. In *Usenix Annual Technical Conference*, pages 71–82, 2015.