

# XLL: Cross-Layer Logging for Data Deduplication in Consensus-Based Storage

Submission #1710

## Abstract

Modern distributed storage systems exhibit *cross-layer data duplication*, writing data to disk once during a consensus phase and again during a local database logging phase. The result is poor performance and significant write amplification. To remedy this cross-layer redundancy, we design and implement Cross-Layer Log (XLL), a shared log built upon the principle of key-value separation. We use XLL to deduplicate updates within a distributed key-value store (TiKV), leading to a 5.5x increase in write throughput while significantly reducing write amplification by 73%. We also demonstrate the effectiveness of our crash recovery protocol in maintaining data integrity.

## 1 Introduction

Distributed storage systems are notoriously complex [17, 49, 50, 55]. The distributed database CockroachDB [50] is implemented in 477K lines of Go; the distributed file system Ceph [55] has 1.6 million lines of C++ code; the database Postgres [49] has 1.6 million lines of C code. Along with the interface visible to clients, each of these systems implements complex distributed consensus protocols, transaction management, fault tolerance machinery, and numerous durability mechanisms.

One way to manage complexity in large systems is through classic systems construction techniques: *layering* and *abstraction* [38]. By dividing the large-scale system into well-defined subsystems, the complexity of building, testing, and maintaining each layer is significantly reduced [12].

Unsurprisingly, layering has costs. For example, log-on-log [56] and journal-on-journal [24, 25] performance problems arise when log-structured application writes are layered atop log-structured filesystems [46], journals, or flash FTLs [19].

In this paper, we first identify a problem found in many distributed storage systems that persist data in separately constructed consensus and storage layers, which we term *cross-layer data duplication*. This problem arises when systems first log data during the execution of the consensus protocol (e.g., Paxos [28], Raft [36]) to ensure correct replication despite node failures; then, when writing the data to disk in a local key-value store upon each node (e.g., RocksDB [1]), the data is written to disk again when applying the commands of the consensus state machine. Cross-layer data duplication produces multiple unnecessary internal disk writes for each logical write, reducing scalability for write-intensive workloads and increasing write amplification.

To remedy this problem, we design and implement *Cross-Layer Log (XLL)*, a shared storage library. XLL extends ideas found in key-value separation systems [5, 6, 23, 26, 30, 53];

however, existing techniques are not flexible enough to share values between multiple layers of a system. With XLL, a system can log commands safely to disk during consensus, but critically keep a reference to persisted data; later, a local database can utilize this reference during its logging phase, recording just the reference to disk instead of rewriting the entire data payload. XLL thus enables data payloads to be written to disk once instead of twice, increasing performance and reducing write amplification significantly.

To show the benefits of the XLL approach, we integrate it into TiKV [51], a production-quality distributed key value store. TiKV is built using Raft for consensus and RocksDB for persistence; we modify the system to store data into a cross-layer log, and then utilize the reference in both the consensus and data subsystems.

XLL is realized with three novel techniques. The first is *coordination-free write sharing*. This technique extends nameless writes [58], in which the storage library returns the byte-offset of the write, by utilizing the temporal flow of data through application layers to avoid cross-layer coordination. The second is *compiler-assisted targeted reads*. Multi-layer systems store data payloads in serialized messages. To avoid extra overheads while reading data from XLL, it is vital to reference specific data within each serialized message. The third is a *consensus-directed crash recovery* protocol. We develop a two-phase recovery protocol using data stored in XLL, first replaying a bounded segment of Raft commands from the log, then restoring the state of the database before resuming operation.

Through detailed evaluation, we show that a deduplicated write path improves overall write throughput by up to 5.5x on a write-only workload compared to TiKV, our baseline system. This represents an additional performance benefit over a version of the system which solely separates keys and values without cross-layer data duplication. Write amplification in our deduplicated system is 73% lower than the baseline system, due to the elimination of redundant write-ahead logging, while read throughput, an improvement over the baseline, matches traditional key-value separated designs. Lastly, we demonstrate the effectiveness of our crash recovery protocol by inserting synthetic failures in key parts of the system’s critical write path.

The rest of this paper is organized as follows: we discuss the organization of layers in consensus-based storage systems and introduce the cross-layer data duplication problem (§2). We then discuss why well-known deduplication techniques are not well-suited to solving the problem, and identify several challenges in doing so (§3). We describe our implementation of XLL, an append-only log shared library, special XLL extensions to RocksDB, and its integration with TiKV, a

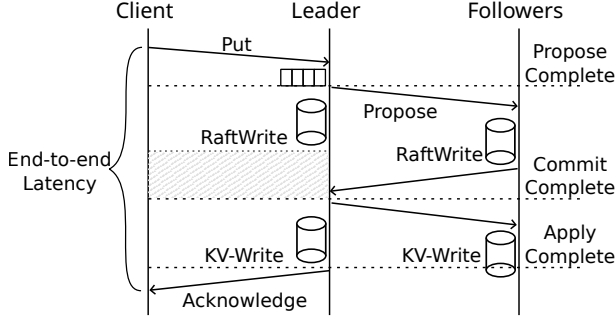


Figure 1: The critical path of a single client `Put` request is divided into `Propose`, `Commit`, and `Apply` phases. Each phase involves communication and disk writes.

consensus-based distributed key value store (§4). Lastly, we finish with an evaluation of the benefits of our approach (§5).

## 2 Background and Motivation

Highly-available, fault-tolerant, and consistent distributed storage systems are often built using replicated state machines. With the popular replicated state machine model [29, 47], distributed nodes reach consensus, or agreement, on commands before applying the command to their own local state machine. In practice, replicated state machines are deployed as multiple layers that are independently responsible for the two tasks: first, consensus for consistently ordering operations, and, second, storage for durably recording the state or data. Consensus is usually achieved with protocols such as Raft [37] or Paxos [28], which maintain a persistent *consensus log* on disk. The storage layer of the system is the materialized view of the state machine.

For simplicity, in this paper we focus on a distributed storage system that implements and exports a key-value store: the commands sent by clients are `Put` and `Get` operations on keys in the store. For the consensus layer, we adopt the terminology of the Raft protocol [37].

Figure 1 shows a simplified model for a `Put` operation across nodes in a distributed storage system. The consensus runtime (e.g., Raft) on the leader node receives commands (`Put`) either from a client or another node in the system. Upon receiving a command, the leader persistently appends the command to a consensus log (`RaftWrite`) and interacts with multiple follower nodes to *commit* the command (the exact steps vary by consensus protocol). After the leader has been notified that the commit is complete, commands are sent to the state machine runtime, where the command is deserialized and executed, or *applied*; applying the commands involves another disk write (`KV-Write`). While not necessary for correctness, to ensure read-after-write consistency many systems wait until the apply phase is complete before responding to the client. Thus, from the perspective of the client, the end-to-end latency of a command includes communication time and execution of all phases of the consensus protocol, including the apply phase. Thus, without further optimizations [16, 34], clients typically wait for two disk write operations.

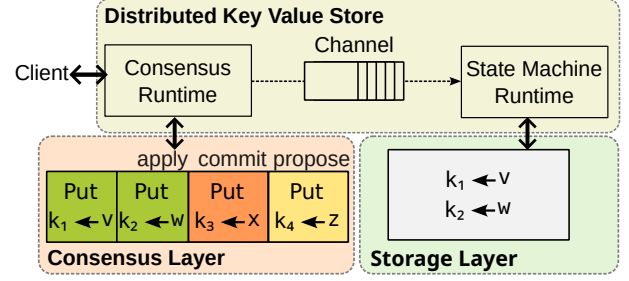


Figure 2: Data in a distributed key-value store is processed in multiple runtimes and storage layers. Committed commands are sent asynchronously via a message channel to the state machine runtime where they are applied.

Figure 2 focuses on the two layers and their respective operations within a single node of the distributed storage system. The consensus runtime layer is responsible for appending commands to a persistent *consensus log*; operations in the log are tracked via *index* as either proposed, committed, or applied (e.g., the `Put` of value  $v$  to key  $k_1$  has been applied, but the `Put` of value  $x$  to key  $k_3$  has only been committed). The state machine runtime is responsible for applying committed commands within the local storage layer (e.g., the values for keys  $k_1$  and  $k_2$  are appropriately updated). Thus, the two applied commands and their data are duplicated on disk, appearing in slightly different forms in each layer.

Table 1 shows that state-of-the-art distributed storage systems, such as TiKV, CockroachDB, etcd, Spanner, and Cassandra [11, 15, 21, 27, 50], do indeed separate consensus and storage into distinct layers. While different systems use different consensus protocols and messaging protocols, each system uses either two separate storage libraries or two distinct instances of the same library for the consensus log versus storage. These libraries are either standalone key-value storage systems like RocksDB, Pebble and BoltDB, or structured file formats that store key-value mappings like Tablets and SSTables.

We describe the consensus and storage layers in more detail.

### 2.1 Consensus Layer

Consensus systems such as Raft [37] and Paxos [28] maintain a persistent consensus log on disk, prioritizing data safety and availability over speed [3]. After receiving a command, nodes write it to the consensus log and vote over a series of epochs until a quorum is reached. If a quorum is found, the command is considered committed and nodes must apply the command to the underlying state machine. Commands are linearizable [20], meaning concurrent updates are serialized as each achieves consensus. Linearizable semantics on a distributed key-value store make it an attractive platform on which to build more complex transactional storage systems [4, 40, 50].

Commands must be persisted before a node may vote on them. For persistence, some systems use their own log implementation (e.g., Etcd), but many systems store the consensus log in a key-value store (e.g., TiKV, Cockroach). In these latter systems, keys correspond to the position of the

System	Consensus	C-Log / Storage	Message
TiKV	Raft	RocksDB/RocksDB	Protobuf
Cockroach	Raft	Pebble/Pebble	Protobuf
Etd	Raft	CLog/BoltDB	Protobuf
Spanner	Paxos	Tablet/Tablet	Protobuf*
Cassandra	Gossip	CommitLog/SSTable	Custom

Table 1: Consensus, storage and message serialization methods used in distributed storage systems.

command in the log (i.e., its index number), and the values are the commands. The keys in the consensus log are used internally and not exposed to clients.

## 2.2 Storage Layer

The storage layer of the system is the materialized view of the state machine. All of the systems in Table 1 use key-value stores to organize data in their storage layer. Cassandra and Spanner manage key-value pairs in structured files like SSTables and tablets, while TiKV, CockroachDB, and etcd use a self-contained key-value store (i.e., LSM trees RocksDB and Pebble, or B-tree BoltDB). For the rest of this paper, we focus on Log-Structured Merge trees, which are popular choices for node-local storage in distributed storage systems [2, 13, 31], specifically on RocksDB.

**RocksDB:** RocksDB has a multi-threaded write path and strong durability guarantees. When key-value pairs are written to RocksDB, the data is written to an on-disk write-ahead log (WAL) [33] and inserted into an in-memory skiplist, known as a memtable. Once the memtable reaches a threshold size, it is made immutable and flushed to disk as a sorted-string table (SSTable). SSTables store the persistent data in the LSM tree in successively larger levels. Flushed memtables make up the highest and smallest level of the LSM tree (level 0). In case of a crash or power outage where unflushed data is still in memory, RocksDB uses its WAL for data recovery. Writes to the WAL, while essential for protection against data loss, limit the throughput of the system because writing to disk is much slower than memory.

Writes to the WAL and level 0 of the LSM tree form the foreground write path of RocksDB. Every key-value pair must be written twice – once immediately to the WAL, and once more when the memtable is flushed. When levels becomes full of SSTables, RocksDB asynchronously runs a compaction routine which consolidates data in one level into larger SSTables stored in the next level, gradually pushing old data further into the tree and deleting stale data. Compactions form the background write path of RocksDB. Compactions contribute to total write amplification, but are scheduled internally by RocksDB, not as an immediate response to a key being inserted into the system.

When a key is read, RocksDB reads from the highest level, starting with the memtable. Keys not found in memory are searched for on-disk starting with level 0. When a key is overwritten, the old value is not removed until a compaction operation. By reading from higher levels of storage first, reads

are guaranteed to return the most recent version of a key. If a memtable were to be irrecoverably lost (e.g., if not backed by a WAL), on reboot the system would revert back to an earlier state because the most recent writes were only memory-durable.

## 2.3 Deduplicating Consensus and Storage

Distributed storage systems thus write similar, but not identical, data to both the consensus layer and the storage layer within each node. This double writing has two problems. First, writing data to disk twice results in significant write amplification, requiring more local storage capacity and consuming more local disk bandwidth. Second, as shown in Figure 1, writing data twice harms the request end-to-end latency observed by clients, since both disk writes occur on the critical path and must be performed synchronously.

Existing generic deduplication methods are not well-suited to deduplicating cross-layer data within a distributed storage system. Offline deduplication, while effective for reducing space consumption, does not reduce foreground write activity; for this reason, offline deduplication is more suitable for archival and backup systems. Online filesystem deduplication [14, 48], in which deduplication occurs on the critical write path, unacceptably lengthens end-to-end write latency, particularly for synchronous writes [57]; these filesystems incur high computational cost to identify duplicate blocks via block fingerprinting and significant memory overhead to maintain the fingerprint index.

Previous application-level optimizations have focused on eliminating extra copies from the data path. PASV [22] identifies double writing *within* MyRocks, a MySQL database built on top of RocksDB, between the database binlog and RocksDB write-ahead log. While removing a write-ahead log eliminates one unnecessary write in the system, it does not remove data duplication problem across different layers. An implementation goal of the SMARTER [8] Paxos-driven storage system was to minimize the number of critical path writes; the authors observe that a single critical path write would be possible using log-structured storage, but they did not explore this solution.

## 3 Deduplicating Cross-layer Data

In this section, we describe our approach for eliminating cross-layer data duplication with XLL. We discuss our design goals and examine three approaches for data organization in TiKV, a representative production-level distributed key-value store. Finally, we describe three important challenges for deduplication and our novel solutions for addressing each of them.

### 3.1 Design goals

Our cross-layer deduplication system for replicated consensus storage systems has the following goals.

**1. Ease of Use.** The use of deduplication should be transparent to clients; there should be no changes to the crash-recovery or data consistency semantics of the service. While it is acceptable to require some modifications to the original storage system to enable deduplication, its use should not be intrusive

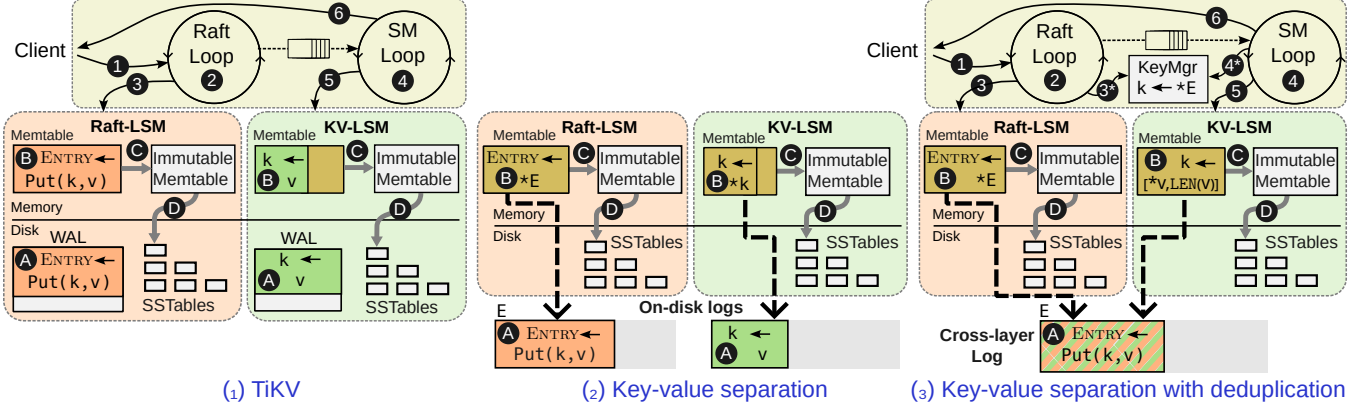


Figure 3: Each figure shows Raft and State Machine (SM) runtimes and associated storage on a single node. (1) uses separate LSM trees for Raft and KV storage. (2) stores values in logs external to each LSM tree. (3) is a layer-deduplicated system using XLL as a single external log. The two LSM trees in (3) contain references to data in XLL – \* denotes byte-offset within log.

or prevent other system-specific optimizations (e.g., operation batching and application-native serialized data formats).

**2. Minimal disk usage.** Performing fewer writes is beneficial for both saving disk capacity and improving throughput. We note that cross-layer duplication is only one source of write amplification: our approach should combine well with other techniques for removing redundant writes of data *within* each layer, such as removing write-ahead logs.

**3. Fast read and write paths.** To fully realize performance benefits, the system should avoid extra computation; our approach should not require extra work to identify duplicate data in the write path or to deserialize data in the read path.

### 3.2 Consensus and Storage in TiKV

In this paper, we use TiKV, a production-quality distributed key-value store, as our representative storage system. Figure 3 shows how TiKV can be progressively restructured so that its two LSM trees can efficiently refer to deduplicated data.

nFigure 3 part (1) describes the original version TiKV. TiKV uses Raft [37] as its consensus layer and persists Raft and State Machine data in two separate LSM trees (marked as Raft-LSM and KV-LSM). Each layer (Raft and State Machine) is a separate runtime within TiKV, implemented internally as an asynchronous event loop. Steps ①–⑥ describe the flow of data through each runtime in the system, while steps ①–④ are internal steps within each LSM tree.

When a client request ① arrives in the Raft layer, it is serialized ② to a protocol buffer and written to Raft-LSM ③; its log index (ENTRY in the figure) is used as its key. Asynchronously, after the command is committed, the Raft loop sends a message to the state machine loop via a channel. The state machine runtime deserializes and parses the command ④, executes it (i.e. writes to KV-LSM) ⑤ and sends an acknowledgement to the client ⑥.

Internally to each LSM tree, data is written in the foreground to both the write-ahead log (WAL) and in-memory memtable (steps ① and ②). As memtables become full, they are converted to immutable memtables ③ and written to disk

as SSTables ④. Serialized Raft commands are written to Raft-LSM in step ③ using their Raft entry number as a key. During step ⑤, when the command is applied by the state machine event loop,  $v$  is written to KV-LSM using the key  $k$  from the command. Thus, in the original version of TiKV, forms of the key and value are duplicated across both layers and ultimately written to disk four times – twice to the two WALs, and twice more to SSTables in each LSM tree.

In Figure 3 part (2), we consider an intermediate version that separates keys and values in each LSM tree and effectively eliminates the WALs. This version of the system is akin to running TiKV on a key-value store like Wiskey [30] or Titan [53]. When used within TiKV, the runtime steps ①–⑥ are the same, so we omit them for brevity. The main difference is that, for each LSM tree, step ①, writes the data to an external log rather than the WAL; the memtable is then modified to contain a reference to the location of the data in the log in step ②, e.g.  $*E$  and  $*k$ . Steps ③–④ are the same. Key and value separation has many performance benefits including reducing the size of the LSM tree, improving write performance, lowering compaction overhead and write amplification, and improving read performance due to better caching of the tree [30]; however, key-value separation does not resolve the cross-layer duplication issue.

Figure 3 part (3) shows a new optimized system with cross-layer deduplication, where the on-disk logs from part (2) are merged into a single *cross-layer log*, XLL. While the execution steps for the Raft runtime are the same, the state machine runtime must now know the location of the written data in XLL after step ③. After step ③, we add ③\*, in which the Raft loop writes the location of data,  $*E$ , to an in-memory hashmap called the *KeyManager*; an entry with  $*E$  is added for every key in the Raft command. Prior to writing each key to KV-LSM in step ⑤, the state machine runtime fetches  $*E$  from the *KeyManager* ④\*, and calculates a new location,  $*v$ , the byte-offset of value  $v$  within XLL. Note that there is no step ⑤A. The SM loop writes the location and size of  $v$ , which are sufficient information to read  $v$  from XLL, to KV-LSM without

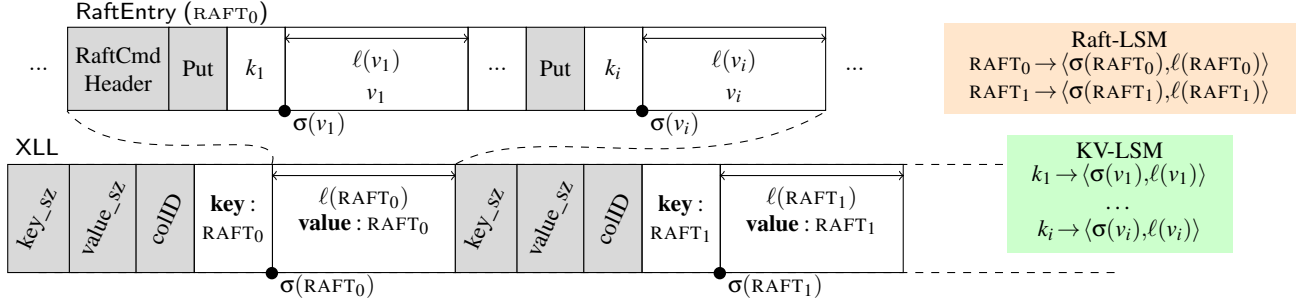


Figure 4: A detailed view of the XLL log and a Raft entry stored within. On-disk metadata is shown in gray. Values referenced from either LSM tree are shown in white.  $\sigma(x)$  refers to the *offset* of  $x$ , while  $\ell(x)$  refers to the *length* of  $x$ . The tables show the contents of each LSM tree given the state of the log.

an additional log write in (5B). The remaining steps are identical to the earlier systems. Thus, the data is written only a single time to XLL, which is then referenced by both LSM trees.

### 3.3 Challenges

There are three fundamental challenges that must be solved to design a system that leverages a cross-layer log. Each of these challenges presents an opportunity to design an efficient mechanism, respectively, for writes, reads, and crash consistency.

**1. Cross-layer data referencing.** The locations of data written to XLL (such as \*E) must be shared with each layer of the system.

**2. Targeted reads.** The state-machine loop must be able to calculate \*v, a location in a serialized buffer, without extra deserialization overhead.

**3. Efficient crash consistency.** Data stored in KV-LSM is memory-durable between memtable flushes. A recovery protocol which does not depend on additional write-ahead logging is necessary to recover data in KV-LSM.

### 3.4 Solutions

We describe solutions for handling these problems efficiently with three techniques: coordination-free write sharing, compiler-assisted targeted reads, and consensus-directed crash recovery.

#### 3.4.1 Coordination-free write sharing

Not only must each TiKV runtime be able to determine the location of data within the log, but writes to the cross-layer log must be ordered. In layered storage systems like TiKV, data accesses follow a temporal flow: the Raft runtime writes to the cross-layer log, and the state-machine runtime later references those writes. To efficiently share write locations and avoid contention, we introduce *nameless appends* an extension of nameless writes [58].

In a nameless write, clients submit write requests, and the storage device or library returns the location of the data after it has been written. The cross-layer log supports *nameless appends*, requiring a small amount of internal state to track the head of the log. Given the temporal flow between Raft and the state machine, nameless writes provide a natural

mechanism for *coordination-free sharing*. Once the Raft layer publishes the data location in the *KeyManager*, no additional coordination is needed to reserve space in the log, nor must the Raft layer maintain sequencing state for log writes.

The application manages log offsets received in response to Raft writes in the *KeyManager* hashmap. As Raft entries are written to Raft-LSM and the cross-layer log, the Raft entry keys and log offsets are added to *KeyManager*. Entries may have more than one command. Later, when the commands are applied, the offset for the entry holding the command is retrieved from the *KeyManager* using the command index to reconstruct the entry key. After all commands in the entry have been applied, its offset is removed from the hashmap.

#### 3.4.2 Compiler-assisted targeted reads

Reads (i.e., Gets) must be serviced quickly within the key-value store. TiKV provides strong consistency via leader read [52] on keys written with Raft; however, this means key-value data is buried within serialized Raft entries in XLL. We introduce a new technique called *compiler-assisted targeted reads* to support targeted access within serialized Protobufs.

To illustrate the additional complexity of accessing data in the log, Figure 4 shows a segment of XLL in detail: the log segment has two serialized Raft entries, RAFT<sub>0</sub> and RAFT<sub>1</sub>. Raft entries are written to the log with XLL metadata describing the key size, value size, and LSM column ID. The top portion of the figure shows the contents of RAFT<sub>0</sub>, a Raft entry containing multiple Put operations along with their keys and values. Raft entry metadata has been simplified and serialization headers (for example, variable-width integers denoting serialized field length) have been omitted for clarity.

Consider the steps for reading  $v_1$  from RAFT<sub>0</sub>. Serialized formats like Protobuf do not support random reads because string and byte fields (along with the integers encoding their length) are variably sized. Thus, Protobufs traditionally require full deserialization before data can be read. Naively, reading  $v_1$  requires reading RAFT<sub>0</sub> fully, deserializing the entire Protobuf, and extracting  $v_1$  from within. Unfortunately, this results in extra computational overheads for reads that are not present in layer-duplicated systems that store values

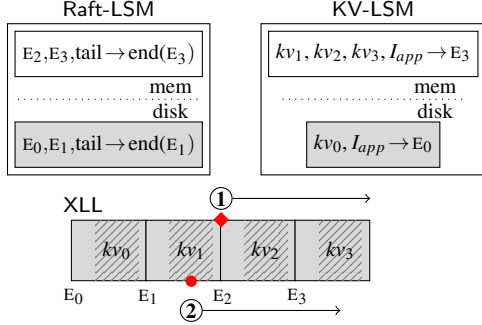


Figure 5: Recovery procedure. Phase ① of recovery loads persisted Raft log entries back into Raft-LSM. Phase ② loads memory-durable key-value pairs into KV-LSM.

separately from the consensus log.

*Compiler-assisted targeted reads* enable targeted access within serialized Protobufs. The Protobuf compiler generates serialization and deserialization code for Protobuf messages defined in a .proto file. It also generates a language-native structure with fields corresponding to the fields in the Protobuf definition, and accessor and modifier methods to access fields in the struct. When a Protobuf is deserialized, a new structure is allocated in memory, its data fields are populated with deserialized data, and accessor methods may be used to access its data. Our modifications to the Rust-Protobuf compiler generate an additional offset (or vector of offsets for repeated fields) data member to track the byte-offset of each string and byte field in the protobuf during the deserialization process. We access this new data member with a generated `get_FIELD_offset()` method for each string and byte field in the Protobuf.

Raft entries are deserialized within the State Machine runtime as part of command execution. Without performing an additional deserialization, we use the offsets of data fields within the Raft entry to infer the byte offsets for each value within the cross-layer log. For example, the location of  $v_1$ , or  $\sigma(v_1)$  in Figure 4 is  $\sigma(\text{RAFT}_0)$  plus the byte offset of  $v_1$  within  $\text{RAFT}_0$ , which can be found with the generated method `get_v1_offset`. When writing  $k_1$  to KV-LSM, we set its value to a *locator*, a 16-byte concatenation of the corresponding value’s offset and length in the log, shown in the figure as  $\sigma(v_1)$  and  $\ell(v_1)$  (the length is trivially known at write-time). On reads, the locator is retrieved from the LSM tree and used to read the value directly from the log.

### 3.4.3 Consensus-directed crash recovery

Deduplicating data across the Raft and storage layers must not harm data durability or crash consistency. For Raft consensus to operate correctly, nodes must not be able to vote multiple times in the same term and must not lose entries that have been committed; this requires that each node must persist its current term, vote, and log entries [36]; TiKV with XLL provides identical durability guarantees. After any crash, our system recovers successfully regardless of whether a particular Raft entry was applied or whether the applied data in the storage layer is disk durable; this guarantee is achieved with two-phase

*consensus-directed crash recovery* that successively restores data to each LSM tree.

We consider the durability and data safety properties for data stored within XLL. Raft consensus operates correctly as long as earlier entries in its log are persisted correctly, given that a later entry is successfully committed. To guarantee this, each Raft entry is appended to XLL and `fsync` is called prior to it being inserted in the Raft LSM tree. If a crash occurs between the write and `fsync` and only some entries are persisted, the system relies on the local file system property of prefix consistency, which ensures that there are no invalid (partial or corrupted) Raft entries before persistent valid entries; prefix consistency is provided by many modern filesystems [41], including ext4 in ordered mode. Checksums can trivially be added to guarantee data integrity of entries following the valid prefix, but this is not needed for correctness.

Following the initial write of a Raft entry, the data in the entry can always be recovered. Figure 5 illustrates consensus-directed crash recovery when some data is only memory durable in each LSM tree. The figure shows four Raft entries  $E_0, \dots, E_3$ ; each entry has been inserted into Raft-LSM, but only  $E_0$  and  $E_1$  have been flushed to disk. Each Raft entry stores key-value data, which has been applied and stored in KV-LSM. KV-LSM also tracks  $I_{app}$ , the last *applied index*, which is flushed atomically with key-value data; in the figure, in memory  $I_{app}$  is set to  $E_3$ , as all entries have been applied, while on disk, there is an older value of  $I_{app}$  which is  $E_0$ .

After a crash, the system initiates the two-phase recovery process. Figure 5 illustrates these steps for a system with some state in memory and some persisted to disk; on a crash, in-memory data is lost from each LSM tree. In the first phase, commands from the Raft-LSM are replayed. As an optimization, to avoid replaying the entire log, the offset of the most recently stored log entry is tracked in *tail*. On reboot, Raft-LSM starts with  $E_0$  and  $E_1$ , and *tail* points to the end of  $E_1$ . Raft-LSM replays the log starting from  $E_2$  and recovers the next two Raft commands. The headers of each entry contain its size and are used to scan the log.

In the second phase, the state of the KV-LSM is restored. The system first reads the key for each entry from Raft-LSM; these keys are then placed within the in-memory hashmap used for writing offsets of data to the KV-LSM tree. Then, commands within entries starting from  $E_1$  are re-executed (because the recovered  $I_{app}$  is  $E_0$ ) and  $kv_1, \dots, kv_3$  are restored to KV-LSM. By flushing  $I_{app}$  atomically with the key-value data, Raft entries are only applied once, even following a crash.

## 4 Cross-Layer Log (XLL)

XLL is a shared library which is designed to be linked with storage applications. To deduplicate cross-layer data within TiKV, XLL is linked with TiKV and RocksDB during compilation. At runtime, a single instance of XLL is instantiated in TiKV and registered with both instances of RocksDB. In this section, we describe the interfaces of XLL as well as several

### XLL Raw API

```
off_t Append(string data)
int Get(off_t offset, string* data, size_t len)
off_t Head()
int Sync()
Iter NewIter(), Seek(offset), Next(), Valid()
```

XLL RocksDB Extensions	Usage
RegisterXLL(w)	Use XLL object w for all external log operations.
off_t PutExternal(k, v)	Write v to log, $k \rightarrow \langle \sigma(v), \ell(v) \rangle$ in LSM tree. Return $\sigma(v)$ .
string GetExternal(k)	Return v stored in XLL.
DBIterator*	Return a DB iterator to read values from XLL.
XLLDBIterator()	

Table 2: XLL Interface

implementation subtleties for integrating XLL with RocksDB.

## 4.1 XLL Interface and Usage

Table 2 shows the raw interface for XLL and RocksDB extensions for XLL, which are available after an instance of XLL has been registered with RocksDB using `RegisterXLL()`. The extensions, which are used by TiKV and not exposed to clients, use the raw interface internally. XLL could be integrated into more storage applications either by using the raw interface directly, or by developing new sets of extensions for other storage libraries (e.g., Pebble, MySQL binlog).

The XLL raw interface supports nameless appends (`Append`) and byte-offset reads of arbitrary length (`Get`). `Sync` may be called to force log durability after appends. The XLL iterator moves through the log sequentially using XLL metadata to advance the iterator. Sequential log scans are used during recovery 3.4.3.

The semantics of `PutExternal` and `GetExternal` are the same as normal `Put` and `Get` in RocksDB, but the values stored in the LSM tree are 16-byte XLL *locators* – 8-byte offset and 8-byte length used to reference data in the XLL log. Like the standard `DBIterator`, `XLLDBIterator` iterates over keys in the LSM tree using `seek`, `next`, and `prev`, but instead reads the value for each key from XLL.

## 4.2 Implementation Subtleties

We describe how XLL handles batch writes and group commit, range scans, and garbage collection.

### 4.2.1 Batch Writes and Group Commits

To avoid contention at the head of the XLL log, `PutExternal` is implemented with careful consideration of RocksDB’s multi-threaded write path. Although RocksDB supports `Put` and `Get` on single keys, batches are commonly written (even single-key `Put` operations are internally converted to batches of one element). Since XLL writes return the location of each value written to the LSM tree, batches are supported by returning vectors of offsets from RocksDB.

Accommodating optimized group commit operations is non-trivial. RocksDB, to reduce the number of small

synchronous writes to the WAL, uses group commit operations to coalesce multiple writer threads. In RocksDB, waiting writers are added to a waiting list (via lock-free compare and exchange); when a write completes, a single writer is selected as group leader which then batches all waiting operations and writes them all to the WAL.

Writes to XLL replace the WAL, but still the optimized group commit mechanism. Offsets returned by XLL during group commit are carefully managed to account for the ordering across writing threads. First, for each write batch, we build an XLL segment (metadata and data for all values in the batch) and track the offset of each value in the segment. While building the XLL segment, we create a new write batch of keys and XLL locators, which is written to the LSM-tree. Second, because writes from multiple threads may be batched together non-deterministically, we maintain a separate offset vector for each writer thread, which is later returned to the caller; without per-thread state, write operations may return empty vectors if their operations are executed by a different thread.

### 4.2.2 Range Scans

RocksDB provides an iterator to perform sequential scans in the LSM tree. Traditional LSM trees store values alongside keys, making this iterator very efficient. `XLLDBIterator` returns values from XLL, requiring an additional read from the log. Since keys and values are not stored together in XLL, this is effectively an extra random read; while random reads have prohibitive cost on spinning disk hard drives, SSDs have good random read performance and internal parallelism. Prior work on key-value separation [30] implements multi-threaded prefetch for range scans; this could also be applied to our work.

### 4.2.3 Garbage Collection

While LSM trees naturally clean deleted and old versions of data with compaction, XLL needs separate garbage collection so that the log does not grow indefinitely. Garbage collection must determine the liveness of data in the log and relocate live data efficiently, without harming crash consistency. Raft implements a simple garbage collection scheme – any Raft entries with an index lower than the last applied index may be truncated from the log. The complication for XLL is that it must additionally check that the data to be truncated is not live within the KV-LSM.

Garbage collection in XLL is done with a sequential scan. Each iteration appends a chunk of valid data at the head of the log, truncates the tail using `fallocate`, deletes unneeded Raft entry keys in Raft-LSM, and updates keys with new references to the log in KV-LSM. To determine whether or not a Raft entry may be garbage collected, the key of the entry is compared to the *last persisted* applied index. Note that this comparison does not require a separate LSM tree lookup, and is simply an integer comparison with the key stored in XLL. It is crucial to compare against the *on-disk* applied index, as more recent updates to the applied index are only memory-durable, and garbage collecting those entries would break crash recovery.

To guarantee the persistence of the applied index, we flush the KV-LSM memtables prior to garbage collection.

Raft entries may be garbage collected if they contain no live data referenced in KV-LSM. Data liveness is determined by parsing the entry and building a list of keys in all `Put` commands in the entry. For each key, we query KV-LSM and verify the offset returned by the query matches the position of the associated value in the entry. If the offsets match, the key matches the most recent version of the key in KV-LSM; therefore, the key is live and we append the entry to the head of the log and update the key with the location of relocated value.

In our current implementation of TiKV-XLL, garbage collection simply runs every two seconds in a background thread. There are numerous heuristics which could be used to reduce this frequency (e.g., wait until the applied index has increased by a minimum amount or until a minimum amount of data has been written to the log). We leave their study for future work.

## 5 Evaluation

We evaluate our deduplicated distributed key-value store, TiKV-XLL, with the following questions in mind:

1. Does TiKV-XLL have lower write amplification than TiKV and other layered storage systems?
2. What impact does XLL have on the end-to-end latency of clients for key-value workloads? How does XLL impact the propose, commit, and apply phases of consensus protocol?
3. Does XLL maintain strong crash consistency semantics?
4. How do application-agnostic methods for data deduplication like XLL compare to application-specific approaches?

TiKV, our baseline system, refers to the publicly-available TiKV 5.4. TiKV-XLL is our deduplicated version of TiKV using XLL. Unless otherwise noted, XLL garbage collection is disabled. To isolate the effect of cross-layer data deduplication, we ablate with a version, TiKV-XLL-SO (Cross-Layer Log Separation Only), that uses two separate instances of XLL, one each for the Raft and storage layers; TiKV-XLL-SO is equivalent to part (2) of Figure 3 and provides the benefits of KV-separation [30, 53]. TiKV-XLL-SO retains the other read and write path optimizations in XLL, including disabling the WAL for both LSM trees, using RocksDB group commit, and reading values directly from the log using locators stored in the LSM tree.

All versions of TiKV range-partition the key space amongst replicas, with each replica acting as a leader for a range-partitioned *region*; regions are periodically split when they grow too large. For TiKV-XLL and TiKV-XLL-SO, we range partition according to the number of keys in the region rather than size, as LSM tree size is a poor measure of region size when values are stored outside the LSM tree.

Experiments are run on five Cloudlab c6615 nodes with Ubuntu 22.04 unless noted. Each has an AMD 9354P

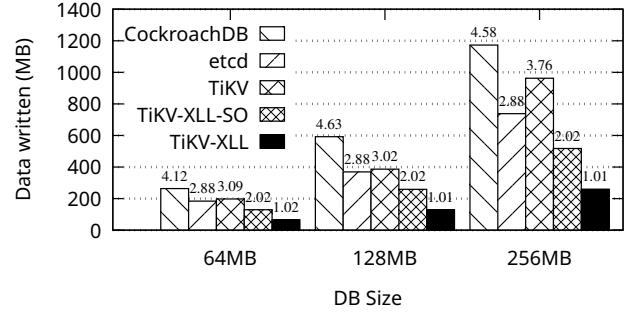


Figure 6: Total bytes written during small database load. Numbers above each column are write amplification.

processor with 32 cores, 192GB memory, and two 800GB NVMe SSD hard drives. The nodes are connected by 100Gb ethernet. We use three nodes as a replicated TiKV cluster to ensure data replication and majority quorum for Raft operations. The other two nodes are configured as benchmark clients, and the TiKV placement driver runs on one of the client nodes. All TiKV data, including XLL, is stored in an ext4-formatted partition on the second hard drive.

### 5.1 Write Amplification

Our first experiments examine the amount of foreground data written across several distributed key-value stores. Reducing the amount of written data is beneficial not only because it can improve performance, but also helps with device lifetime and filesystem aging [42]. In addition to TiKV and our TiKV variants, we also evaluate CockroachDB and EtcD: CockroachDB and TiKV use two LSM trees to store Raft data and committed key-value data, while EtcD uses a custom log for Raft and a persistent B-tree for key-value data.

Figure 6 compares the foreground requests written to disk on a single node (using `strace`); the numbers above each bar show the amount of write amplification. We load small databases of 64MB, 128MB, and 256MB using 1KB records to avoid background compaction. All of the systems except TiKV-XLL have cross-layer data duplication, leading to write amplification of at least 2x. Of these, TiKV-XLL-SO and etcd have the lowest write amplification because they do not perform separate write-ahead logging. In contrast, TiKV-XLL has only 1-2% write overhead for each experiment. Compared to TiKV, TiKV-XLL reduces write amplification by 73%.

### 5.2 Latency of Consensus Phases

We next show that TiKV-XLL significantly improves the per-request latency of write operations relative to TiKV. We use detailed per-request measurements on the leader node to show the cumulative distribution of request latency during the propose, commit, and apply phases of the consensus protocol (matching Figure 1. The time to propose and commit a command the interval between the request arrival at the leader and the commit being complete (i.e., proposal to followers, local log persistence, and notification from the follower nodes).

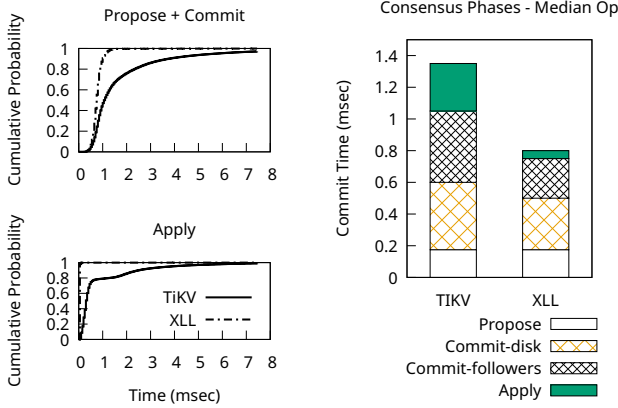


Figure 7: CDF of elapsed time prior to operation commit and apply. The right plot shows a detailed breakdown of TiKV operations during the prepare and commit phases.

Apply is the time between commitment of the request and successful application of the request to the KV-LSM tree.

The two CDF plots on the left side of Figure 7 show the completion time of each phase for 16KB write requests while loading a 50GB database. TiKV-XLL has dramatically faster phases than TiKV: the performance of propose+commit is faster due largely to key-value separation and the elimination of extra write ahead logging; the apply phase is faster because updates to the KV-LSM can simply refer to the persistent XLL written to the Raft-LSM in the propose+commit phase.

The right plot of Figure 7 further breaks down the time per operation. In both systems, the propose phase is dominated by queuing time at the leader and so XLL does not improve latency. The commit phase has two substantial costs: writing to the consensus log on the leader and receiving acknowledgement from a quorum of followers. Although these steps are performed concurrently, in our measurements, acknowledgements were always received after writing to the consensus log; therefore, `Commit-disk` is the time for the consensus log write, and `Commit-followers` is the additional time waiting for a quorum. Consensus log writes on both the leader and followers are made faster due to key-value separation, so end-to-end commit time is improved for the median write operation. The apply phase is faster with XLL because updates to the storage layer (KV-LSM) do not write the complete key-value and simply point to the persistent XLL.

### 5.3 YCSB Performance

Figure 9 compares the throughput of TiKV to TiKV-XLL and TiKV-XLL-SO for the YCSB [10] workloads, normalized to TiKV. We use `go-ycsb` [18] with 64 client threads, which obtained the maximum throughput for each system; memory was restricted to 32GB using `cgroups`. To ensure data is not read from memory, we load a 100GB dataset and drop the page cache between workloads. To show how deduplication affects performance differently depending on record sizes, we run YCSB with 1KB and 16KB records.

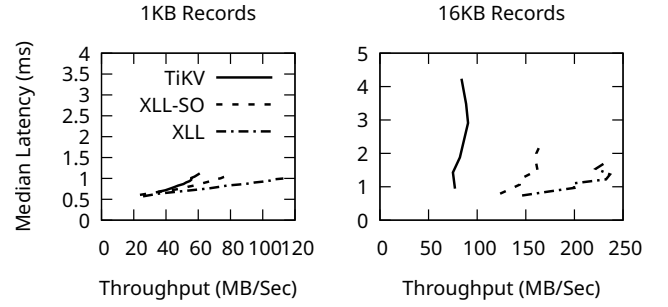


Figure 8: Write operation latency vs. increasing throughput.

Both systems using key-value separation outperform TiKV for all write-intensive workloads (Load, A, F). TiKV-XLL outperforms TiKV by roughly 2.9x for loading 1KB records and roughly 3.2x for loading 16KB records. While key and value separation in TiKV-XLL-SO provides much of the performance benefit, TiKV-XLL shows an additional performance benefit from its efficient single-write design. This is most apparent in write-intensive workloads with large values, as TiKV-XLL has higher absolute write throughput.

Workload E emphasizes the importance of our iterator for range scans (Section 4.2.2). To evaluate this optimization, we implemented a single-threaded iterator in TiKV-XLL without prefetch. While prior work [30] has investigated multi-threaded iterators and prefetching to mitigate the cost of random reads to an external log, we found that a single-threaded iterator outperformed range scans in TiKV. Range scans consist of a seek followed by iteration over sequential keys. Upon further measurement, we found that the performance improvement in YCSB E for TiKV-XLL and TiKV-XLL-SO was due to significantly faster seek operations for the first key in the range.

**Write and Read Throughput vs. Latency:** Cross-layer deduplication is primarily a write optimization. We find that TiKV-XLL improves write performance while keeping read performance the similar to TiKV-XLL-SO. Figure 8 compares write throughput and latency for TiKV, TiKV-XLL-SO, and TiKV-XLL under a 100% write workload (YCSB LOAD of 50GB). We increase the number of client threads in `go-ycsb` to vary throughput, and measured the median latency of write operations. TiKV-XLL has higher maximum throughput and lower latency at the same throughput than either TiKV or TiKV-XLL-SO due to reduced I/O.

Figure 10 shows that uniformly random read-only workloads are not impacted by deduplication, as long as the targeted read optimization is used. To demonstrate the importance of targeted reads, we measure TiKV-XLL-SR (Slow Read), which removes this optimization. XLL-SR has poor read scalability because it reads from the log at the granularity of Raft commands, rather than KV data. Not only must Raft commands be deserialized before returning the data, but since multiple KV pairs are batched within a single Raft command, more I/O must be performed to read the data. We find TiKV-XLL-SR reads 7x more data and consumes 36x

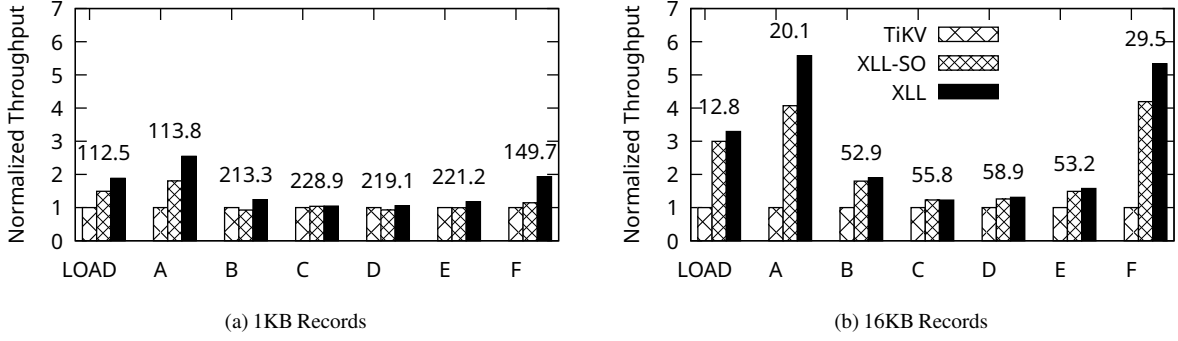


Figure 9: YCSB load and workloads (A: 50% read / 50% update, B: 95% read / 5% update, C: 100% read, D: 95% read-latest / 5% update, E: 95% range / 5% update, F: 50% read-modify-write / 50% read) normalized to TiKV performance. Numbers above each column show TiKV-XLL KOPS/sec for that workload.

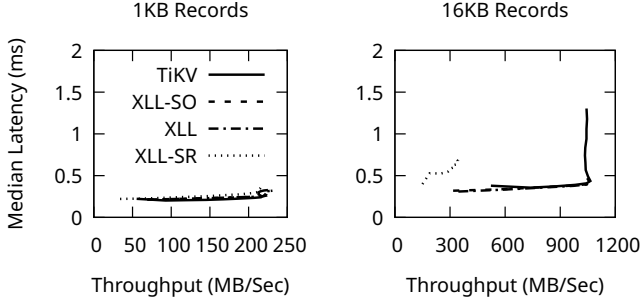


Figure 10: Read throughput in fast distributed configuration (100Gb network, NVMe SSD).

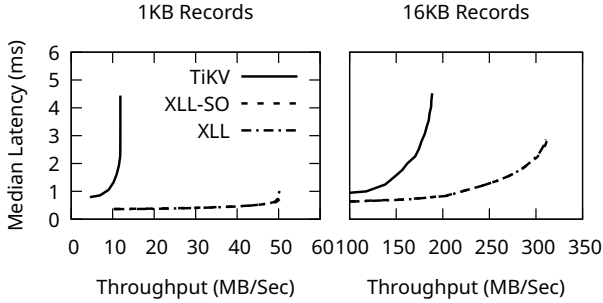


Figure 11: Read throughput on a single node with slower disk.

more CPU time than TiKV-XLL (not shown).

**Read Caching:** Previous systems with key-value separation reported performance advantages for read-heavy workloads because the resulting LSM tree is smaller, improving its caching [30]. However, in our environment with a fast network and NVMe disk, the read bottleneck of TiKV lies elsewhere – reading from memory is not faster than from NVMe. To demonstrate that key-value separation can help read-heavy workloads when the storage system is slower, Figure 11 shows performance with a slower SATA SSD (using only a single node for simplicity). With an SATA SSD, the performance of TiKV is substantially slower than the the KV-separated systems; we note that TiKV-XLL and TiKV-XLL-SO have identical read

paths due to the read optimizations in TiKV-XLL.

Finally, to better understand the write optimizations in TiKV-XLL, we compare TiKV-XLL and TiKV-XLL-SO in mixed workloads with varying proportions of uniformly random writes and reads on a 50GB database. Figure 12 shows that write throughput is significantly better for XLL and read throughput is slight better. Specifically, as the proportion of writes increases, TiKV-XLL has better throughput relative to TiKV-XLL-SO. Because our workload enforces a strict proportion of reads and updates, reducing write bottlenecks in TiKV-XLL also leads to higher read throughput.

## 5.4 Crash Consistency

To evaluate crash consistency, we show that TiKV-XLL matches TiKV’s data recovery for crashes that occur across the write path. Table 3 summarizes the results for injected failures that occur any time between when a client request is written in a Raft entry to its being stored in the storage layer. As desired, commands successfully appended to the Raft log are recoverable in any subsequent failure state.

The original TiKV protocol is as follows. Write requests are durable once appended to the Raft log. Once a quorum of nodes in the Raft group have voted for the entry, it is considered committed. Committed entries are applied to the storage layer. TiKV returns write success to the client only after the entry has been applied to the storage layer, which is relatively conservative since committed entries in Raft are tolerant to node failures. Since any entry that was appended to the leader’s WAL is durable, it can be successfully applied upon system recovery following a crash. TiKV re-applies the entry on reboot even if the system failed before returning to the client.

To evaluate crash consistency, we consider two parts of the write path: first, the write to Raft and second, the write to the storage layer. The former is when the entry becomes durable within the system, and the latter is when the write is exposed to clients of the system. The first three failure points in Table 3 occur during the Raft runtime: Before Raft append is before any data is written to either LSM tree; During Raft append is between the small write of configuration state to the

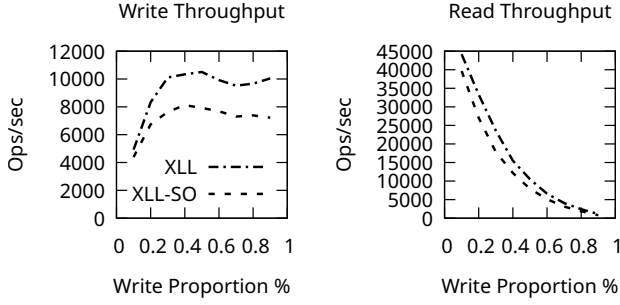


Figure 12: Mixed 16KB update and read workload.

Failpoint	TiKV	TiKV-XLL
Before Raft Append	✓ 10/10	✓ 10/10
During Raft Append	✓ 10/10	✓ 10/10
After Raft Append	✓ + 10/10	✓ + 10/10
Before Apply	✓ + 10/10	✓ + 10/10
Before Callback	✓ + 10/10	✓ + 10/10
After Apply	✓ + 10/10	✓ + 10/10

Table 3: Table of injected crashpoints within TiKV. Checkmarks denote successful read-back of original 100K keys. Plus denotes the same for all additional keys written.

key-value LSM tree and the entry write to the Raft LSM tree; *After Raft append* is after the entry has been logged, but before the execution process has started. The final three failpoints in Table 3 take place during the state machine runtime: *Before Apply* occurs after a command is committed, but before it is applied to storage; *Before Callback* occurs after the write is applied to the storage layer, but before success has been returned to the client; *After Apply* occurs after the write operation is applied and the client is notified. Because XLL removes the WAL from the storage layer instance of RocksDB, applied writes are only memory-durable until the memtable is flushed. By crashing the system immediately after write commands are applied, we rely on Raft to reply committed writes from the log.

We trigger each of the six failpoints multiple times in TiKV and TiKV-XLL. For each failpoint, we load 100K keys, restart the system, and then load an additional randomly-chosen number of keys between 10–60K, at which point a failure is induced. Following the failure, we reboot the system and read back all the keys. We repeat each test 10 times with different numbers of keys for each failpoint.

As summarized in Table 3, TiKV and TiKV-XLL both successfully read back all keys for all failure points after the entry is appended to the Raft log. Before and during the Raft append operations, the new keys may or may not have been added; we verify that the initial 100K keys have not been corrupted and that keys added later are not visible if earlier ones are not. Thus, for this workload, TiKV-XLL achieves the same crash-consistency guarantees as TiKV by recovering data from the Raft log.

## 5.5 Garbage Collection

To determine the maximum performance impact of the XLL garbage collector, we use a write-only workload. We

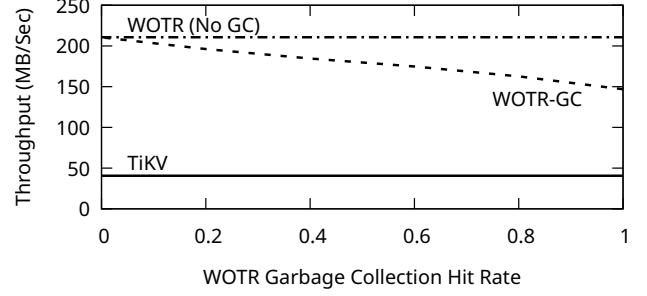


Figure 13: Evaluation of XLL-GC. Write-only workload.

compare the performance of TiKV-XLL and TiKV-XLL-GC using YCSB LOAD (100% write) with 16KB records, which achieved maximum write throughput in previous experiments. During the experiment, the garbage collector runs every two seconds (the TiKV default); we found that adjusting this interval between 1-5 secs had little impact on performance. We simulate garbage collection of different percentages of live data between 0 (all data is invalid and not appended to log) and 1 (all data is valid and appended) by generating a random number for each garbage-collected value; we refer to this as the garbage collection hit rate.

Figure 13 shows the impact of garbage collection on foreground write throughput; we found that write latency remained the same even with the garbage collector. As expected, throughput depends on the percentage of live data, or the hit rate: when there is no live data, then there is no performance impact. When there is live data, writes by the garbage collector to append that valid data do impact throughput as they must lock the head of the log. However, even in the extreme case of garbage collection hit rates of 100%, the write throughput of XLL-GC is well above TiKV.

## 5.6 Comparison with Generic Deduplication

Finally, we investigate generic approaches to deduplication. Across Figures 14 to 16, we quantify the per-write overhead for different chunking methods commonly used in on-line deduplication filesystems, show that even small amounts of overhead have a significant impact on user end-to-end latency. Moreover, these approaches do not identify duplicate data as well as XLL.

To quantify the per-write overheads of different chunking methods often used by on-line deduplicating filesystems, we developed the Low Persistence File System (LPFS) with a FUSE [54]; LPFS absorbs application writes in memory, performs one of several chunking algorithms, and writes the chunk to LevelDB using its hash as the key. LPFS compares three different chunking algorithms: static chunks with a fixed size; Rabin chunks using fingerprinting [35] to identify boundaries and produce variable-sized chunks around a target size (1KB, 2KB, or 4KB); and AppHint chunks using markers inserted by our benchmark to identify potentially duplicated chunks.

Our workloads are modeled on the data written by TiKV. We run two instances of LevelDB on LPFS and insert 10-byte keys and 4KB values (32MB) to each instance; the records across the

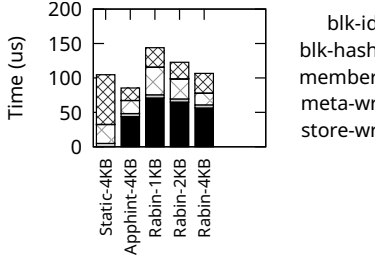


Figure 14: Deduplication costs

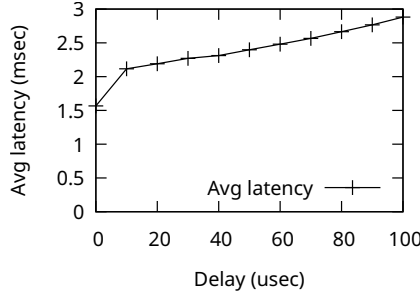


Figure 15: Write latency + synthetic delay

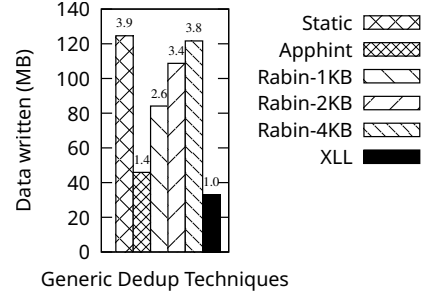


Figure 16: Dedup write amplification

two similar: the values written to the second instance are a concatenation of the key and original 4KB of data. This is a similar write pattern to TiKV and other layered distributed key-value stores, since duplicate data is unaligned across LSM trees.

In Figure 14, we show the cost of each step of deduplication in LPFS. Each write operation is split into five steps: *blk-id* divides the written data into chunks; *blk-hash* computes a hash for its identifier; *member* determines if the chunk is a duplicate of one already in the system; *meta-wr* updates metadata structures on disk; finally, *store-wr* persistently writes the chunk. For identifying blocks, static chunking spends nearly zero time because it only computes a modulo function; however, application hinting is slow because it must scan all input data for substring matching, and Rabin fingerprinting must compute a rolling hash. Overall, each chunking approach adds between 80 (hinting) and 150 (Rabin-1KB)  $\mu$ sec to each write operation.

Figure 15 shows that even a 10  $\mu$ sec per-write delay increases end-to-end latency for clients in TiKV by 500  $\mu$ sec. In these experiments, we add a simple synthetic computational delay to the write path of TiKV, as shown along the x-axis. Realistic delays closer to 100  $\mu$ sec increase client latency by 1.3 ms.

The amount of identified duplicated data for each approach is shown in Figure 16. We see that static chunking does not find duplicate data across LSM trees since the duplicates are shifted by headers and other metadata. Rabin fingerprinting finds more duplication as the chunk size is decreased, but smaller chunks increase its computational cost. Application hinting can find duplicated data, but requires application-specific knowledge of data formats and adds nearly 80  $\mu$ sec to the critical path of each write. In summary, no generic strategy finds duplicates as well as XLL, and XLL has none of the extra costs associated with generic deduplication.

## 6 Related Work

Separating keys and values in LSM trees has been shown to reduce write amplification and improve read and write performance [30]. Other prior work has used LSM trees as an index for accessing larger objects, for example by storing file system metadata within the LSM tree [2, 44]. Our approach to sharing objects between two LSM trees is novel, as far as we know.

Deduplication is often used for backup and archival systems.

Venti [43] identifies duplicate blocks by their hash and stores them in an append-only log. SnapMirror [39] reduces system write latency and bandwidth utilization by mirroring data on the backup server asynchronously. The Data Domain deduplicating filesystem [60] improves write bandwidth by optimizing the index of block hashes, avoiding unnecessary disk lookups in a cache which is too large to fit in memory. LBFS [35] reduces data transfer over the network, identifying duplicate blocks using Rabin fingerprinting.

Primary deduplication identifies duplicates in the critical write path. This can occur within the filesystem [9, 45, 57], or on a block device [59]. Primary deduplication methods incur extra I/O and latency on writes [48]. Deduplication at the filesystem level may be done at block-level or whole-file granularity. Block-level deduplication generally results in a better deduplication ratio [32], but can increase fragmentation on disk [61].

The double logging problem has been well studied in prior work. Log-on-log and journal-on-journal problems can result in garbage collection overhead [56], increased write amplification [24], and extraneous synchronous writes [25]. At the device level, new interfaces [7] have been designed to avoid these resultant behaviors. XLL helps eliminate the double-write problem [22] common to log-on-log systems.

## 7 Conclusion

Consensus-based distributed storage systems persist duplicate copies of data multiple times to disk as data passes through the consensus and storage layers of the system. This cross-layer data duplication arises due to modular construction of these systems, in which separate instances of storage libraries are used for each layer of the system. We have developed an approach to deduplicate data at the application level using a single instance of an append-only log, XLL. We have implemented our approach in a production-quality distributed key-value store, and demonstrate that deduplicating the data path within the application carries numerous benefits including higher overall write throughput and lower write amplification. By managing the locations of data in the log carefully within the application, we develop techniques for efficient read-back of values from the log and strong crash consistency semantics, while eliminating redundant write-ahead logging.

## References

- [1] RocksDB | A persistent key-value store. <http://rocksdb.org/>.
- [2] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. File systems unfit as distributed storage backends: Lessons from 10 years of Ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 353–369, New York, NY, USA, October 2019. Association for Computing Machinery.
- [3] Ramnathan Alagappan, Aishwarya Ganesan, Jing Liu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Fault-Tolerance, Fast and Slow: Exploiting Failure Asynchrony in Distributed Systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 390–408, 2018.
- [4] David F. Bacon, Nathan Bales, Nico Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. Spanner: Becoming a SQL System. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 331–343, New York, NY, USA, May 2017. Association for Computing Machinery.
- [5] BadgerDB authors. Badgerdb. <https://github.com/hypermodeinc/badger>, 2024.
- [6] Vinay Banakar, Kan Wu, Yuvraj Patel, Kimberly Keeton, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wiscsort: External sorting for byte-addressable storage. *Proc. VLDB Endow.*, 16(9):2103–2116, May 2023.
- [7] Matias Björling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 689–703, 2021.
- [8] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos replicated state machines as the basis of a {High-Performance} data store. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.
- [9] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, volume 215, 2003.
- [10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, June 2010. Association for Computing Machinery.
- [11] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, August 2013.
- [12] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, October 1972.
- [13] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *ACM Trans. Storage*, 17(4):26:1–26:32, October 2021.
- [14] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, and Sudipta Sengupta. Primary Data Deduplication—Large Scale Study and System Design. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 285–296, 2012.
- [15] etcd authors. etcd. <https://etcd.io/>.
- [16] Aishwarya Ganesan, Ramnathan Alagappan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Exploiting nil-externality for fast replicated storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 440–456, 2021.
- [17] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Symposium on Operating Systems Principles*, pages 29–43, 2003.
- [18] Go-YCSB authors. go-ycsb. <https://github.com/pingcap/go-ycsb>, September 2024.
- [19] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 127–144, New York, NY, USA, April 2017. Association for Computing Machinery.
- [20] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

- [21] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: A Raft-based HTAP database. *Proc. VLDB Endow.*, 13(12):3072–3084, August 2020.
- [22] Kecheng Huang, Zhaoyan Shen, Zhiping Jia, Zili Shao, and Feng Chen. Removing Double-Logging with Passive Data Persistence in LSM-tree based Relational Databases. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 101–116, 2022.
- [23] George U. Hubbard. Some characteristics of sorting computing systems using random access storage devices. *Commun. ACM*, 6(5):248–255, May 1963.
- [24] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O Stack Optimization for Smartphones. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 309–320, 2013.
- [25] Wook-Hee Kim, Beomseo Nam, Dongil Park, and Youji Won. Resolving Journaling of Journal Anomaly in Android I/O: Multi-Version B-tree with Lazy Split. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 273–285, 2014.
- [26] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. Atlas: Baidu’s key-value storage system for cloud data. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, 2015.
- [27] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review*, 44(2):35–40, 2010.
- [28] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column)* 32, 4 (Whole Number 121, December 2001), pages 51–58, December 2001.
- [29] Butler W. Lampson. How to build a highly available system using consensus. In Gerhard Goos, Juris Hartmanis, Jan Leeuwen, Özalp Babaoğlu, and Keith Marzullo, editors, *Distributed Algorithms*, volume 1151, pages 1–17. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.
- [30] Lanyue Lu, Thanumalayan Sankaranarayanan Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Trans. Storage*, 13(1):5:1–5:28, March 2017.
- [31] Yoshinori Matsunobu, Siying Dong, and Herman Lee. MyRocks: LSM-tree database storage engine serving Facebook’s social graph. *Proceedings of the VLDB Endowment*, 13(12):3217–3230, August 2020.
- [32] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. *ACM Trans. Storage*, 7(4):14:1–14:20, February 2012.
- [33] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992.
- [34] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372, 2013.
- [35] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles, SOSP ’01*, pages 174–187, New York, NY, USA, October 2001. Association for Computing Machinery.
- [36] Diego Ongaro. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, United States – California, 2014.
- [37] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.
- [38] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.
- [39] R. Hugo Patterson and Stephen Manley. SnapMirror: File-System-Based Asynchronous Mirroring for Disaster Recovery. In *Conference on File and Storage Technologies (FAST 02)*, 2002.
- [40] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI’10*, pages 251–264, USA, October 2010. USENIX Association.
- [41] Thanumalayan Sankaranarayanan Pillai, Vijay Chidambaram, Ramnathan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting {Crash-Consistent} Applications. In *11th USENIX Symposium on Operating*

- Systems Design and Implementation (OSDI 14)*, pages 433–448, 2014.
- [42] Jiansheng Qiu, Yanqi Pan, Wen Xia, Xiaojia Huang, Wenjun Wu, Xiangyu Zou, Shiyi Li, and Yu Hua. Light-dedup: A light-weight inline deduplication framework for non-volatile memory file systems. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 101–116, 2023.
  - [43] Sean Quinlan and Sean Dorward. Venti: A New Approach to Archival Data Storage. In *Conference on File and Storage Technologies (FAST 02)*, 2002.
  - [44] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 237–248, November 2014.
  - [45] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *ACM Trans. Storage*, 9(3):9:1–9:32, August 2013.
  - [46] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992.
  - [47] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.
  - [48] Kiran Srinivasan, Timothy Bisson, Garth R Goodson, and Kaladhar Voruganti. iDedup: Latency-aware, Inline Data Deduplication for Primary Storage. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, volume 12, pages 1–14, 2012.
  - [49] Michael Stonebraker and Greg Kemnitz. The POSTGRES next generation database management system. *Communications of the ACM*, 34(10):78–92, October 1991.
  - [50] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20*, pages 1493–1509, New York, NY, USA, May 2020. Association for Computing Machinery.
  - [51] TiKV authors. TiKV. <https://tikv.org/>.
  - [52] TiKV authors. TiKV. <https://tikv.org/blog/lease-read>.
  - [53] Titan authors. Titan. <https://github.com/tikv/titan>, September 2024.
  - [54] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: Performance of User-Space file systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 59–72, 2017.
  - [55] Sage 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 Conference on Operating Systems Design and Implementation (OSDI'06)*, pages 307–320, 2006.
  - [56] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don't Stack Your Log On My Log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*, 2014.
  - [57] Tiangmeng Zhang, Renhui Chen, Zijiang Li, Congming Gao, Chengke Wang, and Jiwu Shu. Design and Implementation of Deduplication on F2FS. *ACM Trans. Storage*, 20(4):21:1–21:50, August 2024.
  - [58] Yiyang Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for flash-based SSDs with nameless writes. In *FAST*, 2012.
  - [59] You Zhou, Qiulin Wu, Fei Wu, Hong Jiang, Jian Zhou, and Changsheng Xie. Remap-SSD: Safely and Efficiently Exploiting SSD Address Remapping to Eliminate Duplicate Writes. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 187–202, 2021.
  - [60] Benjamin Zhu, Kai Li, and R. Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Fast*, volume 8, pages 1–14, 2008.
  - [61] Xiangyu Zou, Jingsong Yuan, Philip Shilane, Wen Xia, Haijun Zhang, and Xuan Wang. The Dilemma between Deduplication and Locality: Can Both be Achieved? In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 171–185, 2021.