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

John Shawger Arnav Jhingran
Andrea Arpaci-Dusseau Remzi Arpaci-Dusseau

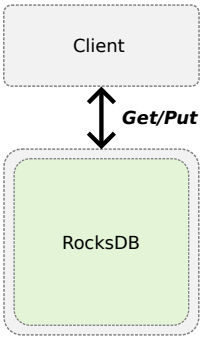
University of Wisconsin – Madison

NSDI, May 2026

Table of Contents

Abstractions Help Us Build Large Systems

Goal: Build a key-value service

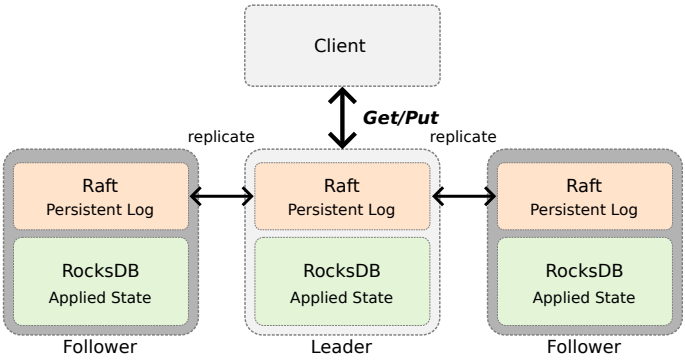


Properties:

- ✓ Key-value interface
- ✓ Persistent
- ✓ Crash consistent

Abstractions Help Us Build Large Systems

Goal: Build a *replicated* key-value service with state-machine replication



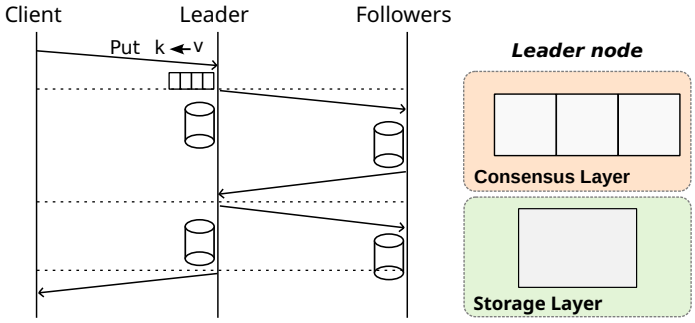
Properties:

- ✓ Key-value interface
- ✓✓ Persistent
- ✓✓ Crash consistent
- ✓ Replicated

Layered systems duplicate functionality

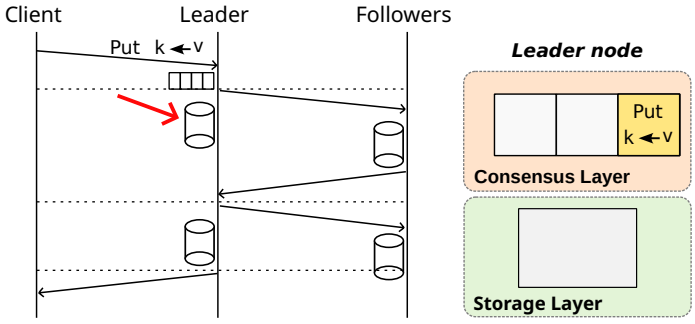
Data Duplication in Consensus and Storage Layers

State machine replication: commands are proposed, committed, and applied



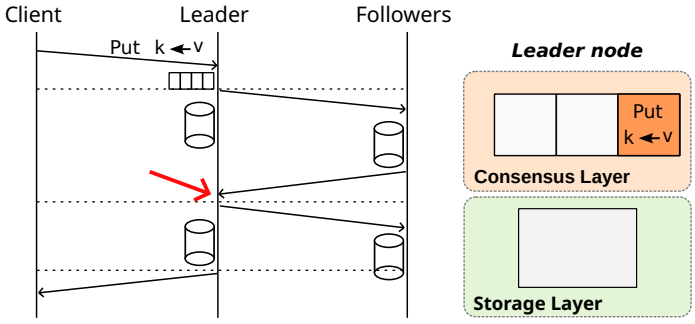
Data Duplication in Consensus and Storage Layers

State machine replication: commands are proposed, committed, and applied



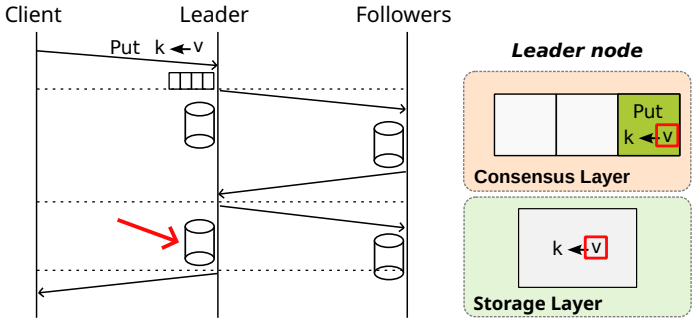
Data Duplication in Consensus and Storage Layers

State machine replication: commands are proposed, committed, and applied



Data Duplication in Consensus and Storage Layers

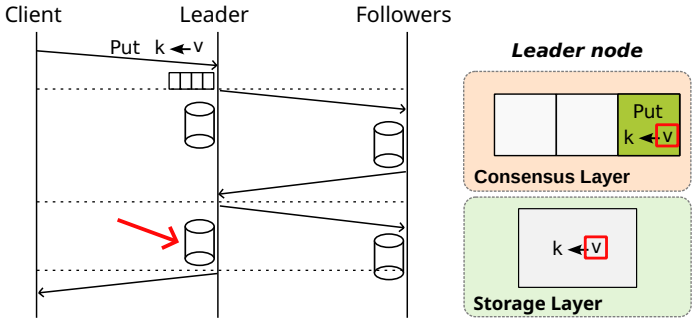
State machine replication: commands are proposed, committed, and applied



- Data payload *v* appears in *both* the consensus log and the KV store
- Multi-layer design is *common*: TiKV, CockroachDB, etcd, Spanner, Cassandra

Data Duplication in Consensus and Storage Layers

State machine replication: commands are proposed, committed, and applied



- Data payload *v* appears in *both* the consensus log and the KV store
- Multi-layer design is *common*: TiKV, CockroachDB, etcd, Spanner, Cassandra

The problem: **Cross-layer data duplication**

Our Contribution

Our solution: Cross-layer logging with XLL

- Cross-layer data duplication
  - Formulate the problem in TiKV, a representative distributed key-value store
  - What would a data deduplicated system look like?

Our Contribution

Our solution: Cross-layer logging with XLL

- Cross-layer data duplication
  - Formulate the problem in TiKV, a representative distributed key-value store
  - What would a data deduplicated system look like?
- Practice
  - XLL: shared library append-only log links to storage application
  - TiKV-XLL: cross-layer data deduplication in a production-quality system

Our Contribution

Our solution: Cross-layer logging with XLL

- Cross-layer data duplication
  - Formulate the problem in TiKV, a representative distributed key-value store
  - What would a data deduplicated system look like?
- Practice
  - XLL: shared library append-only log links to storage application
  - TiKV-XLL: cross-layer data deduplication in a production-quality system
- Evaluation
  - 5.5$\times$ higher write throughput
  - 73% lower write amplification
  - Preserve read performance
  - Preserve persistence and crash recovery semantics

Outline

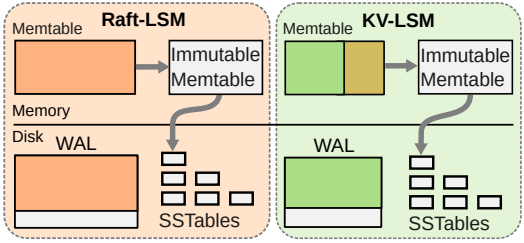
Remainder of the talk

- Rethinking data organization in a replicated persistent key-value store
- High performance in practice
  - Reading from the log
  - Crash recovery
- Evaluation
- Conclusion

Table of Contents

Design: Data organization with cross-layer logging

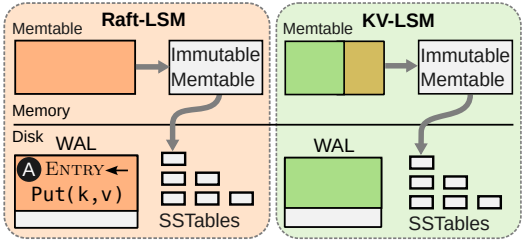
TiKV – Raft and Key-Value Storage



- Raft and KV data in separate LSM trees
- *Raft log entries* written to Raft-LSM
- Applied keys and values written to KV-LSM

Design: Data organization with cross-layer logging

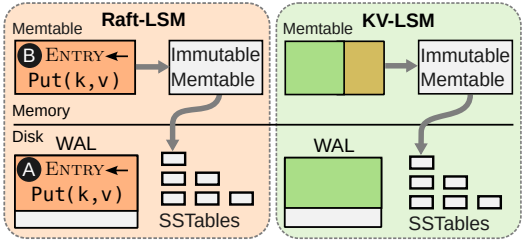
TiKV – Raft and Key-Value Storage



- Raft and KV data in separate LSM trees
- *Raft log entries* written to Raft-LSM
- Applied keys and values written to KV-LSM

Design: Data organization with cross-layer logging

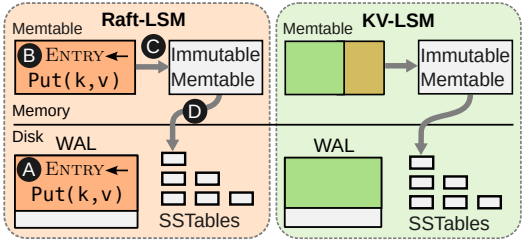
TiKV – Raft and Key-Value Storage



- Raft and KV data in separate LSM trees
- *Raft log entries* written to Raft-LSM
- Applied keys and values written to KV-LSM

Design: Data organization with cross-layer logging

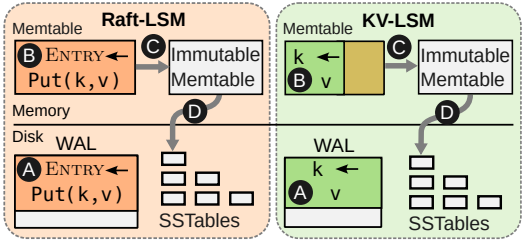
TiKV – Raft and Key-Value Storage



- Raft and KV data in separate LSM trees
- *Raft log entries* written to Raft-LSM
- Applied keys and values written to KV-LSM

Design: Data organization with cross-layer logging

TiKV – Raft and Key-Value Storage

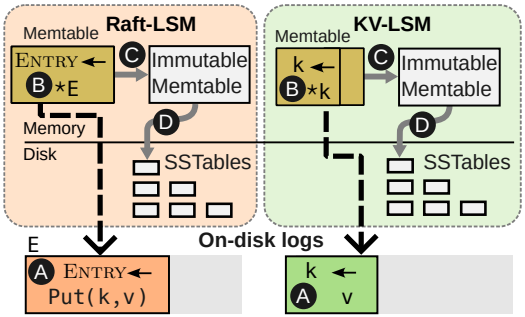


- Raft and KV data in separate LSM trees
- *Raft log entries* written to Raft-LSM
- Applied keys and values written to KV-LSM

Four physical writes for v. Can we write v once?

Design: Data organization with cross-layer logging

Intermediate Step – Key-value separation

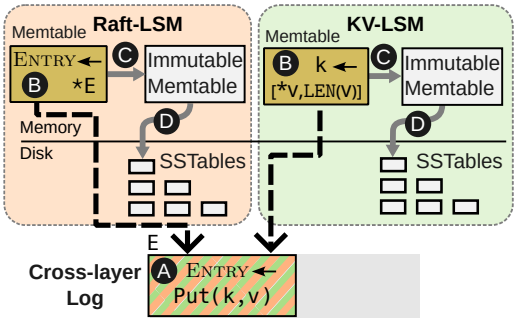


- Data payloads stored in *external logs*
- Keys and references stored in LSM tree
  - Wisckey
  - BadgerDB
  - Titan
- External logs act as WAL
- Key-value separation does not solve cross-layer duplication (v written twice)

- Ⓐ: Persistent write to on-disk log, log byte-offset stored in LSM

Design: Data organization with cross-layer logging

XLL: Cross-Layer Log

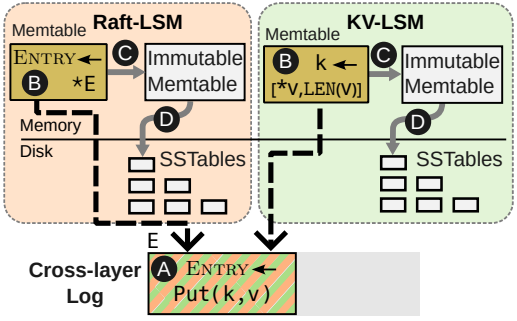


- Store single copy of payload in XLL
- Keys and references stored in LSM tree

- Ⓐ: Persistent write to XLL, reference stored in Raft-LSM (nameless append)
- No Ⓐ for KV-LSM!

Design: Data organization with cross-layer logging

XLL: Cross-Layer Log



- Store single copy of payload in XLL
- Keys and references stored in LSM tree

- Ⓐ: Persistent write to XLL, reference stored in Raft-LSM (nameless append)
- No Ⓐ for KV-LSM!

Problem: *How to reference the payload v?*

Table of Contents

Practical Design Challenges

This talk:

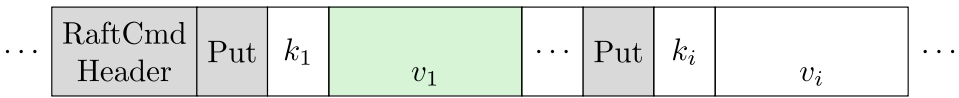
- Efficient reads from XLL
- Crash recovery

More in the paper:

- Append-only writes
- Garbage collection from log
- Range scans

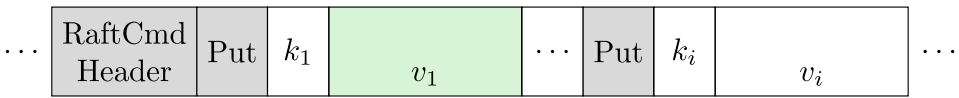
Challenge 1: Efficient Reads

Problem: Distributed system: Raft entries are sent and written in wire format (Protobuf)
Raft entry: Fixed-size header followed by put ops *key* and *value*



Challenge 1: Efficient Reads

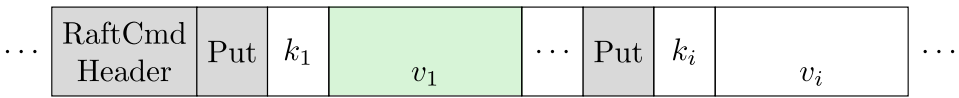
Problem: Distributed system: Raft entries are sent and written in wire format (Protobuf)
Raft entry: Fixed-size header followed by put ops *key* and *value*



Protobufs (and other serialization formats) aren't designed for random access
Variable-width integers encode the length of fields in the buffer

Challenge 1: Efficient Reads

Problem: Distributed system: Raft entries are sent and written in wire format (Protobuf)
Raft entry: Fixed-size header followed by put ops *key* and *value*



Protobufs (and other serialization formats) aren't designed for random access
Variable-width integers encode the length of fields in the buffer

Naive read: read and deserialize entire entry

Challenge 1: Efficient Reads

Problem: Distributed system: Raft entries are sent and written in wire format (Protobuf)
Raft entry: Fixed-size header followed by put ops *key* and *value*

We already parse the entry once to determine the commands

Challenge 1: Efficient Reads

Problem: Distributed system: Raft entries are sent and written in wire format (Protobuf)
Raft entry: Fixed-size header followed by put ops *key* and *value*

We already parse the entry once to determine the commands
Solution: compiler-assisted targeted reads

- For each Protobuf type, the Protobuf compiler generates a struct with access methods. `buf.Parse()` populates the struct

Challenge 1: Efficient Reads

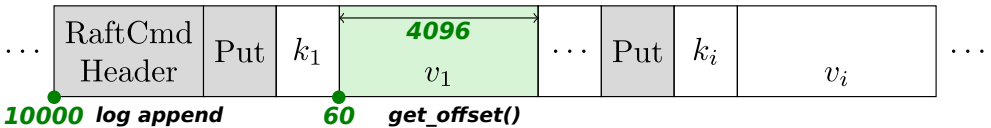
Problem: Distributed system: Raft entries are sent and written in wire format (Protobuf)
Raft entry: Fixed-size header followed by put ops *key* and *value*

We already parse the entry once to determine the commands
Solution: compiler-assisted targeted reads

- For each Protobuf type, the Protobuf compiler generates a struct with access methods. `buf.Parse()` populates the struct
- Generate additional `get_offset()` method for each field

Challenge 1: Efficient Reads

Problem: Distributed system: Raft entries are sent and written in wire format (Protobuf)
Raft entry: Fixed-size header followed by put ops *key* and *value*



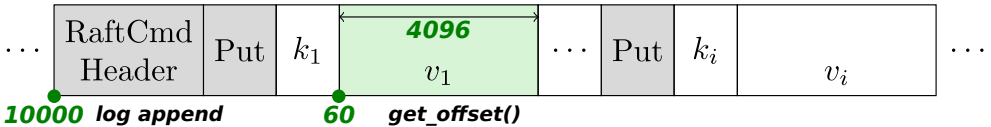
We already parse the entry once to determine the commands

Solution: compiler-assisted targeted reads

- For each Protobuf type, the Protobuf compiler generates a struct with access methods. `buf.Parse()` populates the struct
- Generate additional `get_offset()` method for each field

Challenge 1: Efficient Reads

Problem: Distributed system: Raft entries are sent and written in wire format (Protobuf)
Raft entry: Fixed-size header followed by put ops *key* and *value*



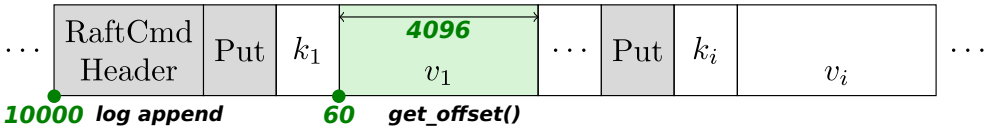
We already parse the entry once to determine the commands

Solution: compiler-assisted targeted reads

- For each Protobuf type, the Protobuf compiler generates a struct with access methods. `buf.Parse()` populates the struct
- Generate additional `get_offset()` method for each field
- Store a *locator* $k_1 \rightarrow \langle 10060, 4096 \rangle$ in KV-LSM

Challenge 1: Efficient Reads

Problem: Distributed system: Raft entries are sent and written in wire format (Protobuf)
Raft entry: Fixed-size header followed by put ops *key* and *value*

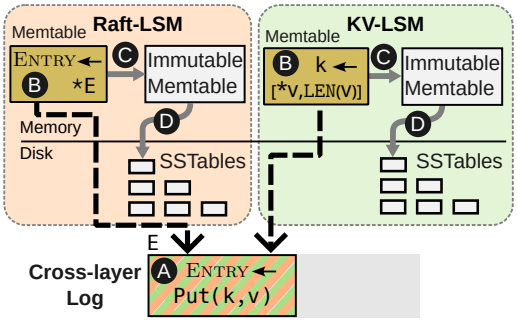


We already parse the entry once to determine the commands
Solution: compiler-assisted targeted reads

- For each Protobuf type, the Protobuf compiler generates a struct with access methods. `buf.Parse()` populates the struct
- Generate additional `get_offset()` method for each field
- Store a *locator* $k_1 \rightarrow \langle 10060, 4096 \rangle$ in KV-LSM

`pread(log, offset, length)` directly reads values from XLL

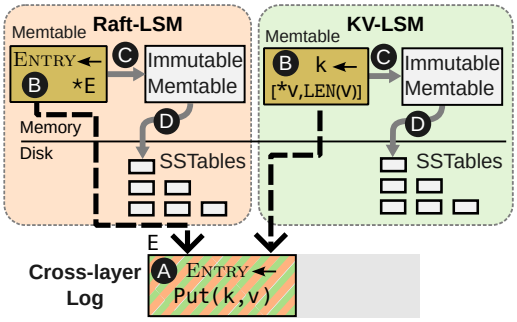
Challenge 2: Persistence and Crash Recovery



No more write-ahead logging!

KV-LSM data is *only* memory-durable between memtable flushes

Challenge 2: Persistence and Crash Recovery

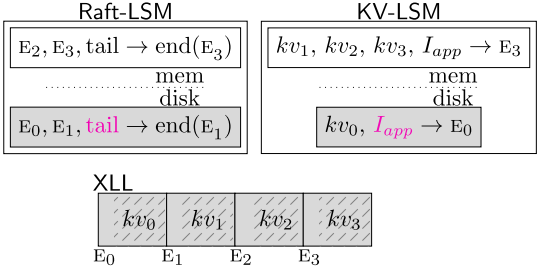


No more write-ahead logging!

KV-LSM data is *only* memory-durable between memtable flushes

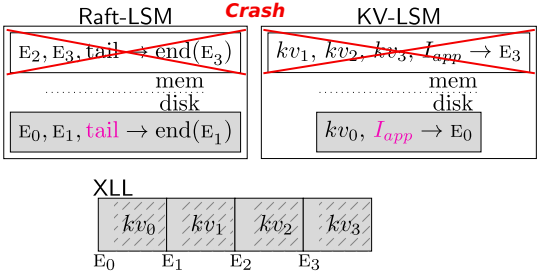
Problem: *How to recover key-value data following a crash?*

Challenge 2: Persistence and Crash Recovery



Consensus-directed crash recovery

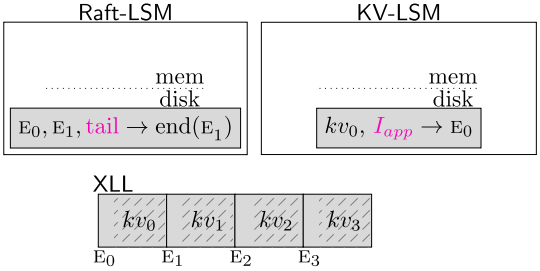
Challenge 2: Persistence and Crash Recovery



Consensus-directed crash recovery

In-memory state lost on crash

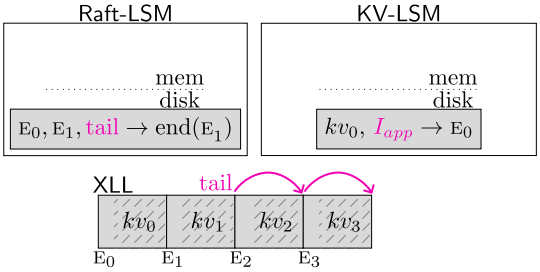
Challenge 2: Persistence and Crash Recovery



Consensus-directed crash recovery

In-memory state lost on crash

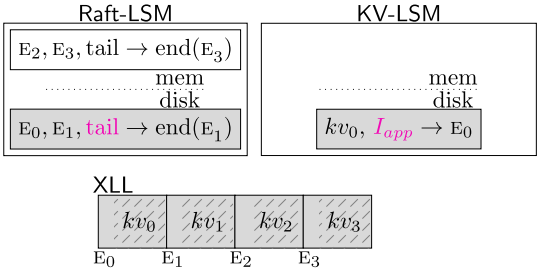
Challenge 2: Persistence and Crash Recovery



Consensus-directed crash recovery

Phase one: use Raft log tail to recover Raft entries

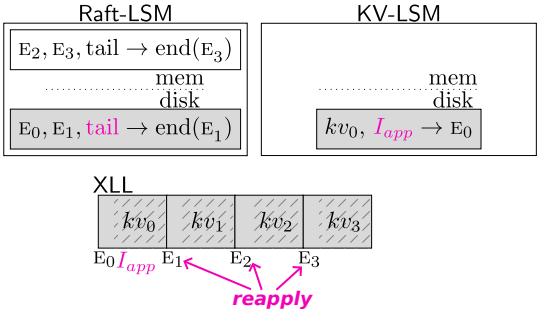
Challenge 2: Persistence and Crash Recovery



Consensus-directed crash recovery

Phase one: use Raft log tail to recover Raft entries

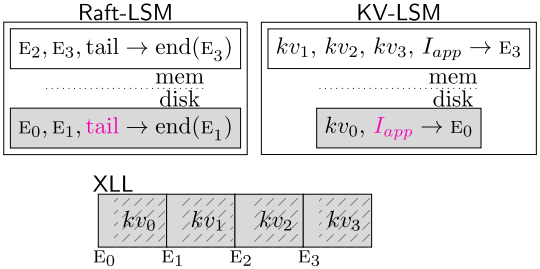
Challenge 2: Persistence and Crash Recovery



Consensus-directed crash recovery

Phase two: re-apply Raft commands from old applied index

Challenge 2: Persistence and Crash Recovery



Consensus-directed crash recovery

Phase two: re-apply Raft commands from old applied index

Table of Contents

Evaluation: Key Questions

- How do XLL writes affect end-to-end latency for consensus operations?
- Does XLL lower write amplification compared to other layered storage systems?
- How does XLL perform in key-value workloads?

(Much more in paper, including garbage collection, crash recovery, read/write scalability)

Evaluation: Experimental Setup

Experiment configuration

Systems:

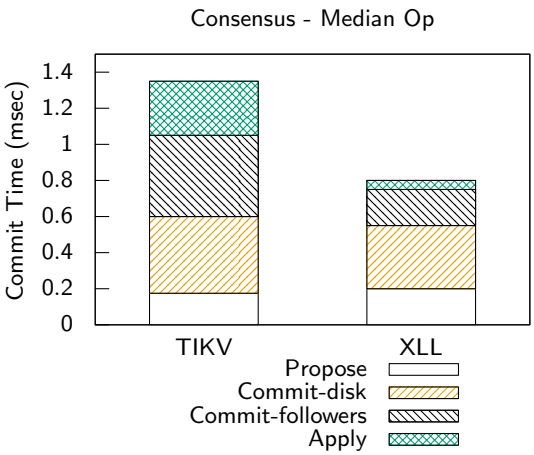
- TiKV
- XLL-SO (Separation Only)
- XLL

Three-node raft cluster:

- 32-core AMD 9354P
- 192GB Memory (KV Service run in 32GB cgroup)
- 800GB NVMe SSD
- 100Gb network

TiKV-SO separates keys from values, but does not deduplicate values

Evaluation: Consensus End-to-End Latency

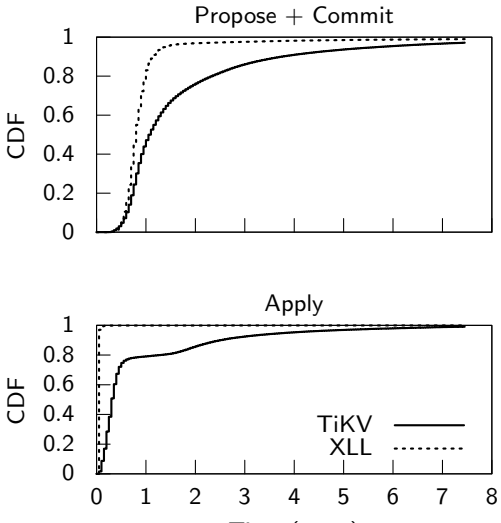


Measure Raft execution phases on leader
16KB writes on a 50GB database

Median op

- **Commit** is faster:
  KV-separation ⇒ quicker disk + quorum
- **Apply** is dramatically faster: KV-LSM just records the locator — no value re-write

Evaluation: Consensus End-to-End Latency

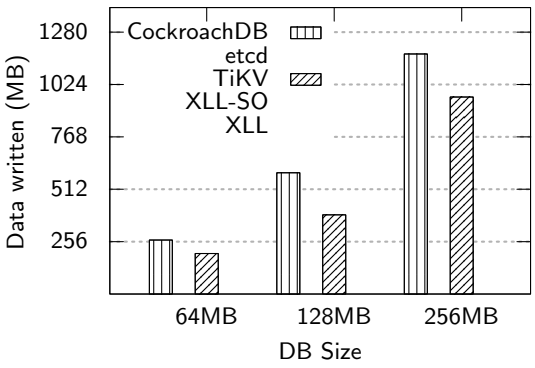


Measure Raft execution phases on leader
16KB writes on a 50GB database

Tail latency

- **Apply** nearly instantaneous

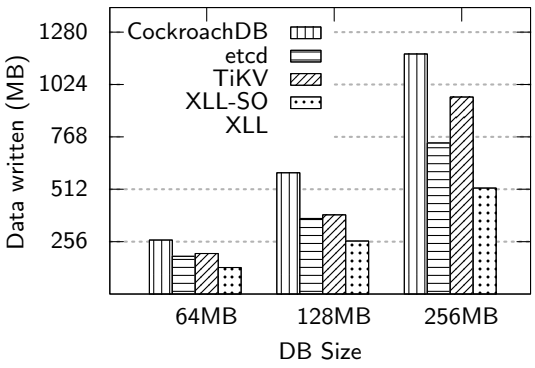
Evaluation: Write Amplification



Foreground bytes written during small-DB loads (1KB records, `strace`)

- TiKV, CockroachDB: 3–4× amplification (2x WAL, 2x LSM)

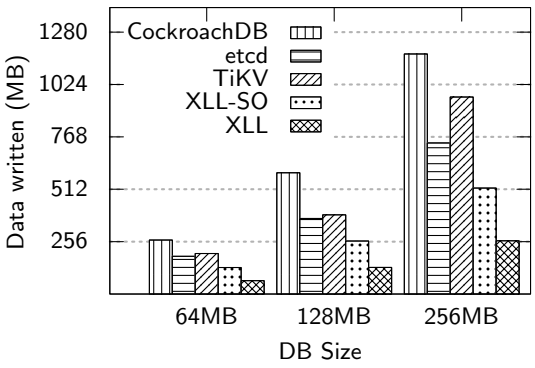
Evaluation: Write Amplification



Foreground bytes written during small-DB loads (1KB records, `strace`)

- TiKV, CockroachDB: 3–4× amplification (2x WAL, 2x LSM)
- XLL-SO & etcd: lower — no WALs, but still double-writes values

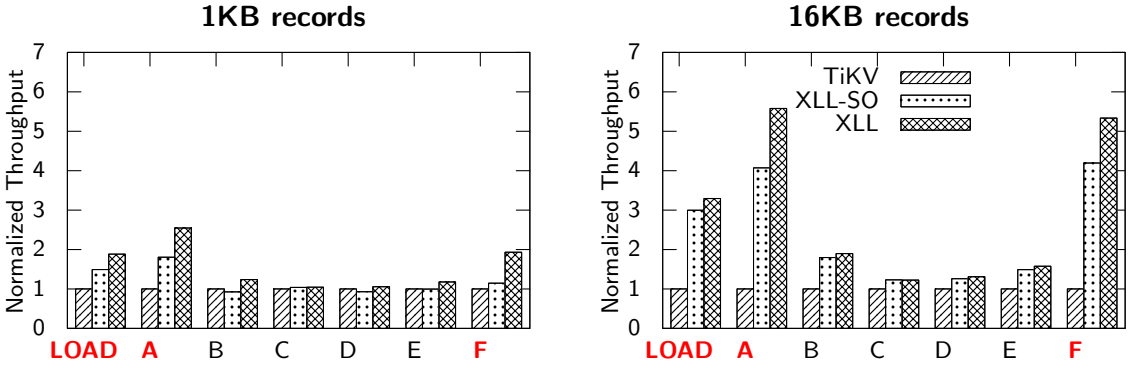
Evaluation: Write Amplification



Foreground bytes written during small-DB loads (1KB records, `strace`)

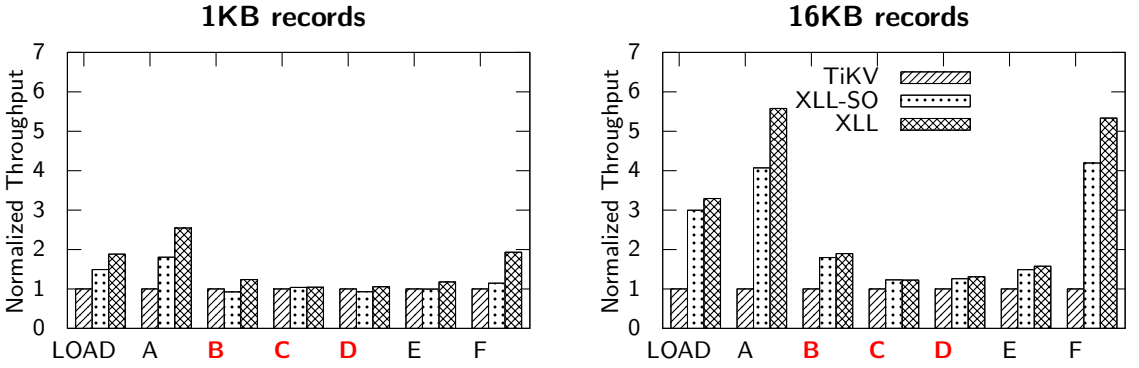
- TiKV, CockroachDB: 3–4× amplification (2x WAL, 2x LSM)
- XLL-SO & etcd: lower — no WALs, but still double-writes values
- **TiKV-XLL: 1–2% overhead**
- 73% reduction vs. TiKV

Evaluation: YCSB



- Throughput normalized to TiKV; 100GB dataset, 32GB page-cache budget
- Write-heavy (Load, A, F): TiKV-XLL up to 5.5× TiKV; clear gap over KV-separation-only
- Read-heavy (B, C, D): matches TiKV-XLL-SO — targeted reads avoid any read penalty

Evaluation: YCSB



- Throughput normalized to TiKV; 100GB dataset, 32GB page-cache budget
- Write-heavy (Load, A, F): TiKV-XLL up to 5.5× TiKV; clear gap over KV-separation-only
- Read-heavy (B, C, D): matches TiKV-XLL-SO — targeted reads avoid any read penalty

Conclusion

Cross-layer data duplication is a pervasive cost in consensus-based storage

XLL – a shared append-only log used by consensus and storage layers

TiKV-XLL: 5.5× write throughput, 73% less write amplification, same crash-consistency guarantees as TiKV

Conclusion

Cross-layer data duplication is a pervasive cost in consensus-based storage

XLL – a shared append-only log used by consensus and storage layers

TiKV-XLL: 5.5× write throughput, 73% less write amplification, same crash-consistency guarantees as TiKV

Strict data isolation between system layers is unnecessary and harmful to performance!

Thank you!