LSM-Raft: Optimizing Raft for LSM-tree Store

XIAOJIAN ZHANG, Tsinghua University, China
XINYU TAN, Timecho Ltd, China
SHAOXU SONG*, Tsinghua University, China
XIANGDONG HUANG, Tsinghua University, China
JIANMIN WANG, Tsinghua University, China

The Raft consensus algorithm ensures strong consistency by replicating a totally ordered log of operations. However, this strict ordering introduces significant redundancy in databases built on Log-Structured Merge (LSM) trees, where follower-side log transmission and compaction often duplicate storage and computation. In this paper, we present LSM-Raft, a generalized extension of Raft that models the replicated log as a sequence of elements—comprising both atomic entries and compacted SSTables. By aligning log semantics and state machine behavior with LSM-tree principles, LSM-Raft enables more compact and efficient replication while preserving correctness. We generalize Raft's core safety properties to accommodate the relaxed log structure and formally verify them using TLA+. Moreover, we propose a cost-aware synchronization strategy that dynamically replaces delayed raw entries with semantically equivalent SSTables, minimizing transmission and compaction overhead. Experimental results show that LSM-Raft improves overall throughput and significantly reduces resource utilization under real-world workloads, demonstrating its practicality in high-ingest, compaction-intensive environments.

CCS Concepts: • **Computer systems organization → Reliability**; • **Information systems → Parallel and distributed DBMSs**; • **Computing methodologies → Distributed algorithms**.

Additional Key Words and Phrases: DBMSs, distributed systems, consensus protocols, LSM-tree, Raft.

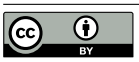
ACM Reference Format:
Xiaojian Zhang, Xinyu Tan, Shaoxu Song, Xiangdong Huang, and Jianmin Wang. 2025. LSM-Raft: Optimizing Raft for LSM-tree Store. *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 340 (December 2025), 27 pages. https://doi.org/10.1145/3769805

1 INTRODUCTION

In distributed systems, ensuring both strong consistency and high performance is a fundamental but often conflicting objective. Consensus protocols like Paxos [27, 28, 31, 33] and Raft [21, 24, 30, 35, 36, 42, 43, 45, 46] provide correctness guarantees but introduce performance overheads, particularly in systems with high write throughput. LSM-tree-based storage engines [15, 32, 34, 39] are widely adopted to mitigate such overheads by optimizing write performance through sequential and batched writes. Due to their write-friendly nature, LSM-trees have become increasingly popular across a broad range of databases [1, 3, 5]. In this paper, we explore how the Raft consensus protocol can be optimized for such LSM-based systems without compromising consistency or fault tolerance.

*Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

Authors' addresses: Xiaojian Zhang, Tsinghua University, China, xj-zhang24@mails.tsinghua.edu.cn; Xinyu Tan, Timecho Ltd, China, xinyu.tan@timecho.com; Shaoxu Song, Tsinghua University, China, sxsong@tsinghua.edu.cn; Xiangdong Huang, Tsinghua University, China, huangxdong@tsinghua.edu.cn; Jianmin Wang, Tsinghua University, China, jimwang@tsinghua.edu.cn.



ACM 2836-6573/2025/12-ART340
https://doi.org/10.1145/3769805

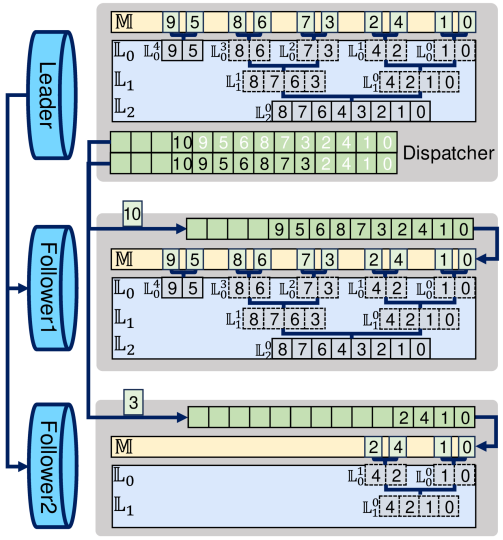


Fig. 1. Motivation of LSM-Raft: redundant flushing and compaction across nodes. In this three-node Raft cluster, the memory region $\mathbb{M}$ contains committed entries 0 through 9. The on-disk region $\mathbb{L}_i$ reflects the LSM-based storage layout. Dashed SSTables have been compacted into solid SSTables on the Leader. While Follower 1 has begun receiving new data, Follower 2 is still synchronizing earlier logs, yet will repeat the same compaction steps as the Leader.

1.1 Motivation

Raft has emerged as a widely adopted consensus protocol due to its understandability and strong correctness guarantees [35, 36, 43] . It ensures consistency by replicating a totally ordered sequence of log entries across all nodes and enforcing deterministic state transitions. Its safety properties—election safety, log matching, leader completeness, and state machine safety— form the foundation for replicated systems requiring fault tolerance [30, 38].

However, the standard Raft design exhibits inefficiencies when applied to LSM-based databases [22, 41]. These databases ingest data at high frequency, periodically flush logs to disk, and merge them into compacted SSTables for long-term storage. Under traditional Raft, each follower replicates raw logs and independently performs identical compaction steps, leading to redundant computation and I/O. Furthermore, when a follower falls behind, it must receive a large backlog of fine-grained entries, even though the leader may have already compacted those entries into a single SSTable.

Such inefficiencies have been observed in Raft-based storage engines like Apache Ratis [24], which treat the log as a flat sequence of atomic entries and miss the semantic equivalence between compacted SSTables and their constituent logs. As a result, they forgo optimization opportunities in modern storage engines that are already compaction-aware.

Moreover, the compaction overhead in LSM trees is non-negligible. Studies show that write amplification can increase significantly when compaction is redundantly executed across all replicas [13, 16, 23, 44]. While advanced compression techniques such as ELF [16, 29] alleviate some of the I/O burden, the fundamental replication semantics of Raft still prevent full reuse of compacted data.

Example 1. *As illustrated in Figure 1, we consider a typical Raft deployment consisting of three nodes. Data points with timestamps 0 through 9 have already been written to memory, replicated across a quorum of nodes, and thus committed. The system is now ingesting the next data point, timestamp 10. Follower 1, which maintains stable connectivity, has received all entries up to index 9, allowing commitment to proceed.*

In contrast, Follower 2 lags behind due to slow log synchronization and has received only entry 3. In such scenarios, Raft continues transmitting raw logs sequentially. However, in LSM-based databases, all flushing and compaction operations performed on the leader will be redundantly executed on followers. This includes the creation of SSTables, their merging, and tiered storage reorganization. Consequently, even though the data is semantically equivalent, followers incur duplicated cost, creating performance bottlenecks that Raft was never designed to address.

This motivates our work: optimizing Raft for LSM-based systems to eliminate redundant processing while preserving strong consistency and fault tolerance. We propose a new protocol, **LSM-Raft**, that decouples logical log replication from physical log representation and enables followers to reuse compacted SSTables produced by the leader. This approach improves overall efficiency without sacrificing safety or correctness.

1.2 The LSM-Raft Approach

To address the inefficiencies in Raft over LSM-based storage engines, we propose **LSM-Raft**, a generalized version of Raft that redefines the structure and semantics of the replicated log. In LSM-Raft, the log is no longer a simple sequence of atomic entries. Instead, it consists of generalized elements, each of which may represent either a single raw data point or a compacted SSTable generated through LSM flush and merge.

Each element is annotated with a metadata tuple containing the Raft term, global index, and compaction level. This metadata allows us to maintain global log order while supporting flexible internal structure. Elements may overlap in data range, which reflects the behavior of real LSM trees but does not violate Raft consistency as long as the resulting state is preserved.

We redefine the state machine as an *LSM state function*, which takes the log as input and simulates LSM compaction to compute the final database state. By modeling the state as a sequence of SSTables, we capture the merge behavior of LSM engines and enable correctness reasoning under compaction-aware semantics.

LSM-Raft also supports a flexible log transmission protocol. When followers fall behind, instead of sending individual log entries, the leader may select an SSTable that is semantically equivalent to the missing sequence and transmit it directly. To support this, we introduce the notion of **log equivalence**: two log sequences are equivalent if they produce the same LSM state. This allows us to replace raw logs with compressed SSTables without breaking consistency.

To make this replacement efficient, we define a *cost model* that evaluates the trade-off between transmission size and compaction overhead. We then design a fast replacement decision algorithm that runs in constant time by comparing metadata only. This enables real-time optimization of replication performance in high-ingestion scenarios.

Finally, the LSM-Raft design preserves Raft's fault tolerance and consistency guarantees. We generalize log matching, leader completeness, and state machine safety to work under the LSM model, and we formally verify these properties using TLA+ [9]. Thus, LSM-Raft maintains correctness while achieving greater efficiency in LSM-based systems.

Table 1. Frequently used notations

Symbol	Descriptions
$\mathbb{M}$	the MemTable in LSM
$\mathbb{L}_i$	the ith level in LSM
E_i	the Raft log element(log entry or SSTable)
$\mathbf{L}$	the Raft log
$\mathbf{S}$	the LSM state
Φ	the LSM state function
ρ	the compression ratio of SSTables
C_{trans}	the transmission cost
C_{comp}	the compaction cost
C	the total cost

1.3 Contributions

While LSM and Raft are established techniques, their non-trivial integration in LSM-Raft introduces new abstractions, provable correctness under generalization, and efficiency-driven mechanisms beyond engineering. Our major contributions in this paper are as follows.

(1) Design Novelty: We generalize Raft's log replication to support the transmission of compacted SSTables (Algorithm 1), enabled by formal abstractions such as generalized log elements (Definition 1). This unifies the consensus and storage layers, allowing Raft to replicate both logical operations and physical storage states.

(2) Theoretical Formulation: To preserve safety under the new design, we generalize Raft's core properties—Log Matching, Leader Completeness, and State Machine Safety (Properties 1, 2, 3)—and formally verify them using TLA+ (Section 4.3). It provides a strong theoretical foundation for safely deploying LSM-Raft in systems.

(3) Technical Contribution: We propose a cost-aware synchronization strategy (Algorithm 2) that dynamically selects between raw entries and compacted SSTables based on transmission and compaction cost. This introduces cost sensitivity into consensus logic, enabling real-time, storage-aware decisions at the leader.

(4) Practical Generality: We implement LSM-Raft in both TiDB [7, 22] and Apache IoTDB [1, 41], and evaluate it with real-world workloads, which demonstrate the practical generality of LSM-Raft. Our results show improvements in overall throughput and significant reductions in CPU, compaction and network cost with unaffected fault tolerance.

2 PRELIMINARIES

To facilitate understanding of the LSM-Raft design and its applicability, we present three core areas of background knowledge: the LSM-tree storage engine, the Raft consensus protocol, and the prevailing database systems.

2.1 LSM-Tree Storage Engine

The Log-Structured Merge (LSM) tree [32, 34] is widely adopted in high-ingestion databases [3–5, 41]. Its popularity stems from its write-optimized architecture, which converts random updates into sequential, batched disk operations. This design minimizes write amplification and enables high-throughput ingestion, making it suitable for workloads with sustained write pressure.

The LSM architecture buffers writes in an in-memory MemTable, which is asynchronously flushed to disk as immutable sorted files called SSTables. These files are managed in a tiered layout across multiple levels (L0, L1, L2, ...). New SSTables are first written to Level 0 (L0), where files may overlap in key range. Subsequent levels contain non-overlapping SSTables, each covering a disjoint key range to support efficient lookups and merges.

To maintain system performance, LSM trees use compaction to merge SSTables across levels, reduce redundancy, reclaim storage, and preserve key ordering. For example, when L0 accumulates too many overlapping files, they are compacted into L1 to reduce read amplification. This multi-level compaction process balances write efficiency, read latency, and disk space usage, forming the foundation of LSM-tree behavior. Understanding this hierarchy is crucial when designing log replication protocols that must reflect consistent and deterministic storage states across replicas.

2.2 Raft Consensus Protocol

In contrast to Byzantine fault-tolerant (BFT) protocols [11, 12, 18], Raft [21, 35, 36, 45] follows a crash fault-tolerant (CFT) model and ensure strong consistency in distributed systems. It includes log replication, leader election, and majority-based commit semantics. As long as a majority of nodes are functioning, the protocol guarantees safe and consistent log advancement.

In Raft, each log entry is uniquely identified by a pair: term (the election round in which the leader was elected) and index (its position in the log). During leader election, nodes vote for a candidate whose log is at least as up-to-date as theirs, based on the latest term and index. This ensures that only the most recent and consistent log is replicated. The elected leader then becomes the sole coordinator of client requests and log replication until the next election.

Raft performs log replication as follows: when the leader receives a client request, it appends the operation as a new log entry in its local log and concurrently sends `AppendEntries` RPCs to all followers. Followers either accept and append the entry to their logs or reject if a mismatch is detected. The leader retries until a majority of followers acknowledge the entry. Once replicated to a quorum, the leader commits the entry and applies it to the state machine. This commit step ensures that only entries replicated to a majority are considered durable and visible to reads.

As a variant of Raft [36], LSM-Raft adopts the standard leader-based read model, in which all client reads are served by the leader node. As stated in the extended version [35], Section 6, "Client requests in Raft are processed through the leader." Before responding to a read request, the leader ensures that the data has been committed by a majority of replicas through the Raft consensus process. This read model guarantees data consistency, because clients always observe the latest committed state that reflects a total order of writes.

2.3 System Model and Target Applications

With the increasing demands for large-scale data ingestion and reliable consistency, modern databases are progressively adopting LSM-tree architectures for storage and Raft as the core distributed consensus mechanism. As discussed in previous sections, this combination has become a mainstream design for ensuring both high write performance and fault-tolerant replication.

Several production-grade systems adopt this architecture pattern. For example, TiDB [7, 22] and Apache IoTDB [1, 41] integrate LSM-tree storage engines with Raft-based replication layers. These systems span diverse application domains, including time-series workloads and transactional or hybrid OLTP/OLAP scenarios, where consistency and write-intensive performance are critical.

LSM-Raft is particularly beneficial in scenarios with continuous, high-volume data ingestion. Representative applications include industrial sensor monitoring, telemetry pipelines, and edge computing deployments- environments that demand both high throughput and strongly consistent replication guarantees.

3 LSM-RAFT DESIGN

This section presents the architecture and core design mechanisms of LSM-Raft. Compared to traditional Raft, LSM-Raft extends the definition of logs to support level-based SSTable replication. After log generalization, we mainly focus on log replication, leader election and safety.

3.1 Log Generalization

In traditional Raft [36], the replicated log is a totally ordered sequence of immutable entries, each tagged with a term and a globally unique, increasing index. Each entry represents a discrete command to be applied to the replicated state machine. LSM-Raft extends this notion by allowing each element in the log to represent either a single entry or a compacted SSTable.

Definition 1 (Element). *Let $\mathcal{D}$ be the domain of time-series records. An element E_i is defined as either: (1) a singleton entry: $E_i = \{e\}$, where $e \in \mathcal{D}$ is a log entry; (2) a compacted SSTable: $E_i = \{e_1, ..., e_n\}$ where $e_j.timestamp < e_{j+1}.timestamp$.*

(`index`, `term`, `logSpan`) is annotated with each element as a metadata. `Level` denotes the storage level in LSM-Tree. `LogSpan` is newly added to denotes the indices range that the entry or SSTable covers.

$$\texttt{LogSpan}(E) = \{e.index \quad \forall e \in E\}$$

We clarify that within the same level, each log element uses the same range size, but the index spans are non-overlapping. This design follows the standard structure of level-based SSTables in LSM-trees, where files at the same level cover disjoint key (or index) ranges to support efficient query and compaction operations [32, 34].

Definition 2 (Log). *The replicated log in LSM-Raft is a sequence of elements:*

$$\mathbf{L} = \langle E_1, E_2, \ldots, E_n \rangle$$

where each E_i has a globally unique index, and elements may semantically overlap in their internal coverage (`logSpan`). These overlaps do not violate index ordering and are resolved by compaction-aware consistency logic.

3.2 Log Replication

In LSM-Raft, log replication must tolerate behind followers and then try to sync entries into the folllowers. This is achieved by comparing not only the index and term, but also the *log span* – the range of logical indices a given element covers. A consistency check involves comparing the leader's and follower's terms, log indices, and spans. If the terms match, the follower determines whether the incoming element aligns, overlaps, or conflicts with its own log.

Example 2. *As shown in Figure 2, LSM-Raft reduces both transmission and compaction overhead by enabling the leader to replicate compacted SSTables directly. Importantly, this optimization does not bypass entry-level synchronization or compromise consistency; instead, it selects lower-cost synchronization paths when the system faces I/O or network bottlenecks.*

3.2.1 Consistency Check Algorithm. We present Algorithm 1, which describes the consistency check logic for appending a new element from the leader to the follower log. The algorithm generalizes the Raft-style `preLogIndex` and `preLogTerm` comparison by also considering the *log span* — the complete logical index coverage of a potentially compacted element. It abstracts the notion through the use of `preLogSpan` and `followerLogSpan`, representing the logical coverage of the leader's and follower's elements, respectively.

Algorithm 1 ConsistencyCheck

Require: `preLogTerm`, `preLogIndex`, `preLogSpan`
Require: `followerTerm`, `followerLogIndex`, `followerLogSpan`
Ensure: `ACCEPT`, `REJECT`, or `NEXTINDEX`

```
1:  if preLogTerm == followerTerm then
2:      if preLogIndex == followerLogIndex then
3:          return ACCEPT                                   ▷ Exact match on index
4:      end if
5:      if NoOverlap(preLogSpan, followerLogSpan) then
6:          if max(preLogSpan) < min(followerLogSpan) then
7:              delete follower log at min(followerLogSpan)
8:              return ConsistencyCheck                     ▷ Delete and Retry
9:          else
10:             return NEXTINDEX                            ▷ Follower index behind
11:         end if
12:     end if
13:     if max(preLogSpan) == max(followerLogSpan) then
14:         return ACCEPT                                   ▷ Exact match on SSTable
15:     end if
16:     if max(preLogSpan) < min(followerLogSpan) then
17:         delete follower log at min(followerLogSpan)
18:         return ConsistencyCheck                         ▷ Delete and Retry
19:     end if
20:     if min(preLogSpan) > max(followerLogSpan) then
21:         return REJECT                                   ▷ Follower index behind
22:     end if
23: else if preLogTerm > followerTerm then
24:     return NEXTINDEX                                    ▷ Follower term behind
25: else
26:     return REJECT                                       ▷ Leader is stale
27: end if
```

If spans are aligned at the boundaries, replication proceeds straightforwardly, as shown in Line 3. However, if any replica fails, span corruption and metadata inconsistency may occur—which is a common situation in Raft. As shown in Line 6 and Line 16, when the follower's span is totally or partially ahead of the leader's, Algorithm 1 triggers local deletion in Line 8. Conversely, as shown in Line 9 and Line 20, when the leader's entry is ahead of the follower's, the algorithm updates the `NEXTINDEX` in Line 10, and retries until an exact match is found in Lines 13 and 14. This design ensures that follower logs are correctly updated either by element deletion or by backward traversal of the leader's log, avoiding over-acceptance that could compromise correctness.

As the standard Raft, the leader delivers data to followers synchronously. That is, our LSM-Raft have the same acknowledgment feedback information from followers for each data replication as the vanilla Raft. For example, in Algorithm 1, each follower returns acknowledgment information such as `ACCEPT` or `REJECT`, which is essential for the leader to determine log alignment, detect inconsistencies, and decide whether to send raw log entries or compacted SSTables. This interaction mirrors the original Raft behavior as described in Section 5.3 of the Raft paper [36].

Overall, this mechanism maintains Raft's safety guarantees — including the Log Matching Property and Leader Completeness, which will be generalized later in Section 4. This abstraction forms the foundation for efficient, consistent replication in LSM-based distributed databases.

3.2.2 Index Management. In LSM-Raft, the leader maintains a pair of indices for each follower: `nextIndex` and `matchIndex`. These are updated upon every successful AppendEntries RPC.

`nextIndex`. This index indicates the log index the leader will send next to the follower. It serves as the current replication position. Initially, it is set to the leader's last log index plus one. When entries are successfully appended to a follower, it is updated as:

$$\texttt{nextIndex} \leftarrow \max(E.\texttt{logSpan}) + 1$$

If an inconsistency is detected, `nextIndex` is decremented to retry synchronization with an earlier log element.

`matchIndex`. This index reflects the highest log index the leader knows to be successfully replicated on the follower. Upon successful append, it is updated as:

$$\texttt{matchIndex} \leftarrow \max(E.\texttt{logSpan})$$

This field is crucial for determining when an entry is safely replicated on a quorum.

`commitIndex`. The `commitIndex` represents the highest log index known to be committed across a majority of nodes. In LSM-Raft, a log element is considered committed if all of its covered indices (as described by `logSpan`) are included in the `matchIndex` of a quorum of followers. This ensures that even in the presence of compaction and log rewriting, every committed entry remains durably and consistently replicated.

3.3 Safety

LSM-Raft retains Raft's original safety guarantees, generalized for its richer log model: A candidate can only be elected if its last log covers all committed log indices known to a quorum. Only entries from the leader's current term are committed directly; earlier ones are committed transitively. If two logs share the same index and term, their application up to that point must produce the same LSM state. These extensions are formally proven in Section 4.

Example 3. *As shown in Figure 3, we illustrate the log replication protocol of LSM-Raft, covering leader commit advancement, follower synchronization, leader election, and crash recovery.*

(a) Commit Advancement via Shared LogSpan. At this stage, Replica #4 has just completed synchronization of the compressed SSTable $\mathbb{L}_0^1$, which covers the log span $\{4, 5\}$. Since this SSTable has been replicated to a quorum (e.g., replicas #2 and #4), we can safely commit entries at index 4 and 5, and advance the leader's `commitIndex` *from 2 to 5.*

If the leader crashes and a new election begins, the LSM-Raft election mechanism remains consistent with Raft: a candidate must contain all committed entries. Replica #2, holding atomic entries up to index 5, qualifies. Replica #4 also qualifies, as its $\mathbb{L}_0^1$ semantically covers $\{4, 5\}$.

(b) #4 Becomes Leader: Synchronization by LogSpan overlap. Now suppose replica #4 becomes the new leader. Its last log index is 6, and it initiates replication using `preLogSpan` $= \{4, 5\}$ *from $\mathbb{L}_0^1$.*

- ***Replica #3*** *holds only entry #4, i.e.,* `followerLogSpan` $= \{4\}$*, which is a strict subset of the leader's span. According to Algorithm Line 16, the entry is considered outdated and will be deleted. The leader re-sends $\mathbb{L}_0^1$.*

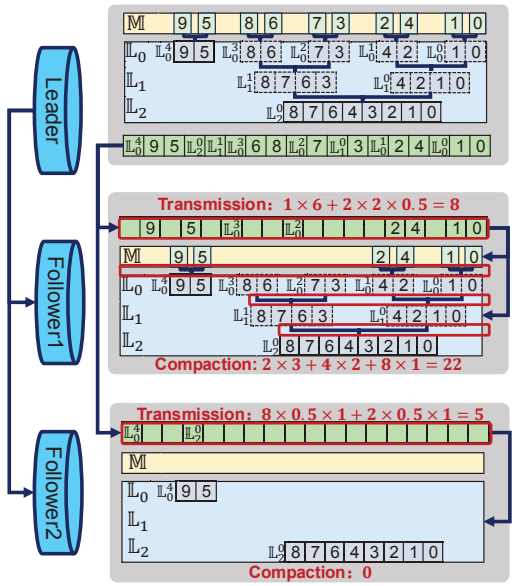


Fig. 2. The leader replicates a mix of atomic entries and compacted SSTables, and selects the transmission granularity based on each follower's state. Follower 1 receives 6 entries (cost = 6) and 2 SSTables (cost = $2 \times 0.5 \times 2 = 2$), totaling a transmission cost of 8, along with a compaction cost of 22 due to redundant local merging. In contrast, since Follower 2 replicates the log too slow, the leader proactively chooses to send 2 SSTables from levels $\mathbb{L}_2$ and $\mathbb{L}_0$, resulting in a lower transmission cost of 5 and avoiding unnecessary compaction.

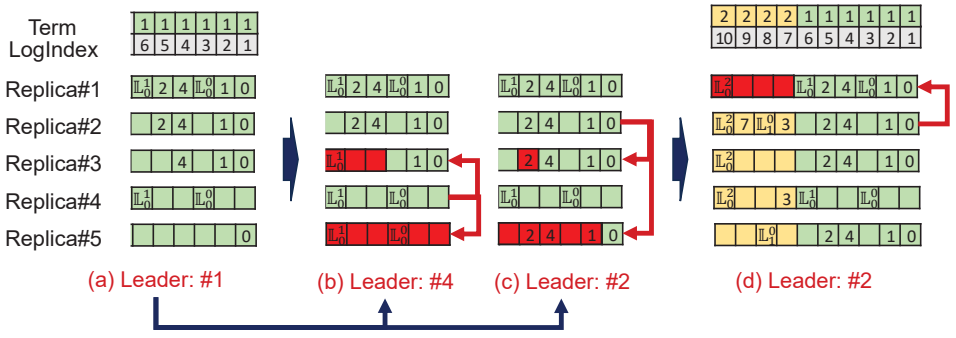


Fig. 3. Example for Log Replication in LSM-Raft

• ***Replica #5*** *holds only index 1, i.e.,* `followerLogSpan` $= \{1\}$*, with no overlap with* $\{4, 5\}$*. According to Algorithm Line 9, it rejects the replication attempt. The leader decrements* `nextIndex` *to 3 and updates* `preLogSpan` $= \{1, 2\}$*. Since there is now an overlap (*$\{1\}$*), the follower deletes its entry at index 1, and then proceeds to synchronize entries from index 3 onward.*

(c) #2 Becomes Leader: Fine-Grained Entry Synchronization. If replica #2 ecomes the leader, its `preLogSpan` $= \{5\}$.

- ***Replica #4*** *holds* $\mathbb{L}_0^1$ *with* `followerLogSpan` $= \{4, 5\}$. *The overlap includes index 5, and it matches the maximum covered index of #2, so according to Algorithm Line 13, the leader considers it up-to-date.*
- ***Replicas #3 and #5*** *are handled as in (b) using* `followerLogSpan` *comparisons, resulting in synchronization of entries with higher index.*

(d) #1 Recovers: Efficient Resynchronization via SSTable. After a stable period, the previously crashed replica #1 restarts and re-joins the cluster. A re-election confirms #2 as the leader. Since #1 is outdated, the leader transmits the compacted SSTable $\mathbb{L}_0^2$ *according to Algorithm Line 23, which semantically covers all missing log entries. This enables replica #1 to synchronize to the latest state in a single transmission, avoiding redundant entry replay and reducing recovery latency and network cost.*

LSM-Raft extends Raft's log matching principle by reasoning over `LogSpan` instead of exact index and term. By comparing leader and follower spans, not only redundant entries can be safely deleted and followers can synchronize via compacted SSTables, but also crash recovery can be accelerated through semantic state transfer. This mechanism significantly reduces replication overhead and improves convergence efficiency in high-ingestion scenarios.

4 CORRECTNESS PROOF

In this section, we present the correctness proof of our LSM-Raft protocol. Unlike the original Raft algorithm which models the state machine log as a strictly ordered sequence of atomic entries, our generalized model allows logs to evolve through compaction, batching, as long as the final state derived from these inputs remains consistent. We organize this section into three parts: state machine definition, property generalization, and formal verification via TLA+.

4.1 LSM State Machine

We define the `LSM State Funtion` as the state machine's final logical output, determined by emulating the behavior of an LSM-tree. The function mainly depends on log and the element order. Although the LSM settings like the SSTable thresholds of different levels in LSM-Tree also affects the final ouput, the different replicas should share the same settings and then we don't consider it.

Definition 3 (LSM State Function). *Given the log* $\mathbf{L}$ *and initial LSM State* $\mathbf{S}$*, the LSM state function generates the new state* $S^{'}$.

$$\mathbf{S}^{'} = \Phi(\mathbf{L}, \mathbf{S}) = \langle E_1^{'}, E_2^{'}, \ldots, E_m^{'} \rangle.$$

where $E_i^{'} = \cup\{E_j, E_{j+1}, ...\}$ *and* $E_j \in \mathbf{L} \cup \mathbf{S}$.

Here, Φ simulates the LSM-tree compaction process through ordered union operation $\cup$. It produces an ordered sequence of SSTables from the log $\mathbf{L}$.

Example 4. ***Out-of-order Entries****: We assume SSTable size thresholds grow exponentially by level [2,4,8,16,…], and use the same setting in the following example. If data arrives in sequential order, such as* $\langle [0], [1], [2] \rangle$*, the state machine compacts it into SSTables as follows:*

$$\Phi(\langle [0], [1], [2] \rangle, \emptyset) = \langle [0, 1], [2] \rangle$$

However, if the data arrives out of order, which is common in industrial scenarios, e.g., $\langle [0], [2], [1] \rangle$*, then after inserting* $[0]$ *and* $[2]$*, the system flushes them together as* $[0, 2]$*. When* $[1]$ *arrives later, it is added separately. The resulting state becomes:*

$$\Phi(\langle [0], [2], [1] \rangle, \emptyset) = \langle [0, 2], [1] \rangle$$

Example 5. ***Duplicate Handling:*** *When a newly added element overlaps with previously ingested ones, the LSM compaction process retains the data from the higher-level SSTable. For example:*

$$\Phi(\langle [0,1,2,3], [4], [4,5] \rangle, \langle [0] \rangle) = \langle [0,1,2,3], [4,5] \rangle$$

Here, the flushed SSTable $[0,1,2,3]$ *from level 2 overwrites the earlier entry* $[0]$ *from level 0 during compaction, since its level is higher. Similarly,* $[4,5]$ *at level 1 dominates* $[4]$ *at level 0.*

When duplicated records with the same key exist across log, we resolve conflicts by preferring the SSTable from higher levels. This strategy follows standard LSM-tree semantics. The rationale is twofold: 1. LSM-based databases typically adopt an append-only write model, where new data is appended sequentially and older entries are eventually compacted or merged. Thus, newer values naturally overwrite older ones during compaction. 2. In cases where updates to historical data do occur, such modifications are either written as newer entries or handled by auxiliary mechanisms. These updates are processed separately from normal compaction logic. As a result, resolving duplicate keys by retaining records from higher levels is both correct and efficient under common LSM design patterns.

4.2 Property Generalization

To ensure that the generalized log semantics still preserve Raft's guarantees, we revisit and adapt Raft's five fundamental properties: Election Safety, Leader Append-Only, Log Matching, Leader Completeness, and State Machine Safety. In LSM-Raft, although log elements may overlap in their time ranges, these overlaps do not affect Election Safety and Leader Append-Only, as both are independent of log contents and rely solely on term and index ordering. We retain their standard definitions. The remaining three properties, however, require generalization to align with the semantics of the LSM-based state machine.

Log Matching. In traditional Raft, if two logs have the same entry at a given index and term, they must be identical in all previous entries. However, with the introduction of SSTables in LSM-Raft, this strict requirement is no longer applicable. Specifically, two logs with different elements could still yield the same LSM state. Therefore, we extend the condition to require that the final state—computed by the LSM state function Φ—be consistent across replicas when log elements match in index and term. The generalized Log Matching as shown:

Property 1. ***Log Matching with Generalization****: if two logs contain an element with the same index and term, then the states of applying all elements up through the given index are equivalent.*

Leader Completeness. In Raft, once an entry is committed in a given term, it must appear in the logs of all future leaders. In LSM-Raft, we generalize this by requiring that all data points contained in a committed element must be included—possibly in compacted form—in future leader logs.

Property 2. ***Leader Completeness with Generalization****: If a log element is committed in a given term, then that all the entries contained in the element will be present in the logs of the leaders for all higher-numbered terms.*

If a log element E_i is committed in term t, then for any higher term $t' > t$, every data entry $e \in E_i$ must be present in some element of the new leader's log:

$$\forall e \in E_i \in \mathbf{L}_t,\ \exists E_j \in \mathbf{L}_{t'} \text{ such that } e \in E_j$$

State Machine Safety. In the presence of asynchronous replication or compaction, some followers may have missing (null) elements at specific indices. When such a follower becomes a leader, it

must ensure consistency by preserving the logical state derived from all non-null elements up to any given committed index. This guarantees eventual convergence, even with differing logs.

Property 3. ***State Machine Safety with Generalization****: if a server has applied a log element at a given index to its state machine, no other server will ever apply a different log element for the same index.*

4.3 Formal Verification via TLA+

To ensure the correctness of LSM-Raft under the generalized log and state machine semantics, we model its behavior using TLA+ [20, 35]. Our goal is to prove that even when the log consists of entries and compacted SSTables, and the state machine follows LSM-tree semantics, Raft's safety properties still hold.

```
-------- MODULE LSMRaft --------
EXTENDS Naturals, Sequences
CONSTANTS
Nodes, Terms, Entries, SSTables, InitialState
VARIABLES
log, committed, state
LogElem ==
[type : {"entry", "sstable"}, content : SUBSET Entries, term : Terms, index : Nat]
LSMState == Seq(SUBSET Entries)
LogPrefix(lg, i) == SubSeq(lg, 1, i)
Phi(logElems) ==
LET flatten == UNION { e.content : e \in logElems }
IN [flatten]
LogMatching ==
\A n1, n2 \in Nodes, i \in Nat :
/\ i <= Len(log[n1]) /\ i <= Len(log[n2])
/\ log[n1][i].index = log[n2][i].index
/\ log[n1][i].term = log[n2][i].term
=>
Phi(LogPrefix(log[n1], i)) =
Phi(LogPrefix(log[n2], i))
LeaderCompleteness ==
\A n1, n2 \in Nodes, i \in Nat :
/\ i <= committed[n1]
=>
\A e \in log[n1][i].content :
\E j \in 1..Len(log[n2]) :
e \in log[n2][j].content
StateMachineSafety ==
\A n1, n2 \in Nodes :
Phi(LogPrefix(log[n1], committed[n1])) =
Phi(LogPrefix(log[n2], committed[n2]))
```

We define the following components in our TLA+ specification. Firstly, a replicated log $\mathbf{L}$ as a sequence of log elements, where each element is either a singleton (entry) or a batch (SSTable). Secondly, the LSM state function $\Phi(\mathbf{L}, S_0)$, which deterministically computes the final state from a log and an initial state. Moreover, properties including Log Matching, Leader Completeness, and State Machine Safety, expressed as invariants over multiple replicas.

The TLA+ specification verifies that for any pair of replicas with matching logs up to a given index (modulo compaction), the resulting state machine outputs remain identical. Additionally, it ensures that once a log element is committed in a term, its effect appears in the logs of all subsequent leaders. A complete listing of our TLA+ module is provided in Appendix [9].

5 LEVEL DETERMINATION MODEL

In the traditional Raft log replication protocol, the leader must transmit each log entry one by one to all follower nodes. However, in real-world systems, some follower nodes may experience synchronization delays due to network congestion or local I/O pressure. This leads to log accumulation on the leader, which impacts write confirmation latency and overall system throughput.

LSM-Raft addresses this problem by proposing an optimization: when a follower falls significantly behind, the leader can transmit a compressed, higher-level SSTable to replace multiple pending raw log entries. This reduces the number of transmission rounds and volume of transferred data, while also relieving the follower of unnecessary compaction overhead.

5.1 Definition of Log Equivalence

We first introduce the notion of **log equivalence**, which determines whether two log sequences result in the same state for a given state machine.

Definition 4 (Log Equivalence). *Let $\mathbf{L}_1$ and $\mathbf{L}_2$ be two log sequences, and $\mathbf{S}_f$ be the current state of a follower. Let $\Phi(\cdot,\cdot)$ be the state function representing the result of applying logs to the LSM-based state machine. If*

$$\Phi(\mathbf{L}_1, \mathbf{S}_f) = \Phi(\mathbf{L}_2, \mathbf{S}_f) \tag{1}$$

then $\mathbf{L}_1$ and $\mathbf{L}_2$ are semantically equivalent under $\mathbf{S}_f$, denoted as:

$$\mathbf{L}_1 \overset{\mathbf{S}_f}{\equiv} \mathbf{L}_2 \tag{2}$$

This equivalence relation allows us to search for lower-cost log variants without compromising final state consistency. It is crucial for LSM-Raft optimization because it guarantees that state machine safaty is maintained, even when the log structure is altered for efficiency.

Example 6. *Consider a Raft cluster with 5 nodes. Each write batch contains 1000 records, and the SSTable size is 10000. After ingesting data $[19001, 20000]$, a level-1 SSTable (index=20) is created and successfully committed by followers 1 and 2. Follower 3, however, is still trying to receive $[10001, 11000]$ of index 11, and also the queued logs from index 12 to 20. A new SSTable covering $[10001, 20000]$ is now available. Let $\mathbf{L}_{v20} = \langle [10001, 11000], \ldots, [19001, 20000] \rangle$ and $E_{sst} = [10001, 20000]$, it easy to derive that $\mathbf{L}_{v20}$ and $\langle E_{sst} \rangle$ is log equivalent. This indicates that sending just E_{sst} instead of all ten original log entries preserves the same final state.*

5.2 Cost Modeling for Log Synchronization

We define a cost function $C(\cdot)$ over the space of log sequences. This function quantifies the overhead incurred when synchronizing logs rom leader to follower. For any log sequence $\mathbf{L}$, the total cost is decomposed as:

$$C(\mathbf{L}) = \sum_{C_i \in \mathcal{D}_C} C_i(\mathbf{L})$$

We focus on two dominant contributors: *transmission cost* and *compaction cost. Transmission cost* reflects the bandwidth and transmission round complexity required to send the logs to the follower. It includes: the sum of entry sizes $|E|$ for raw entries, the compressed size for SSTables (ρ

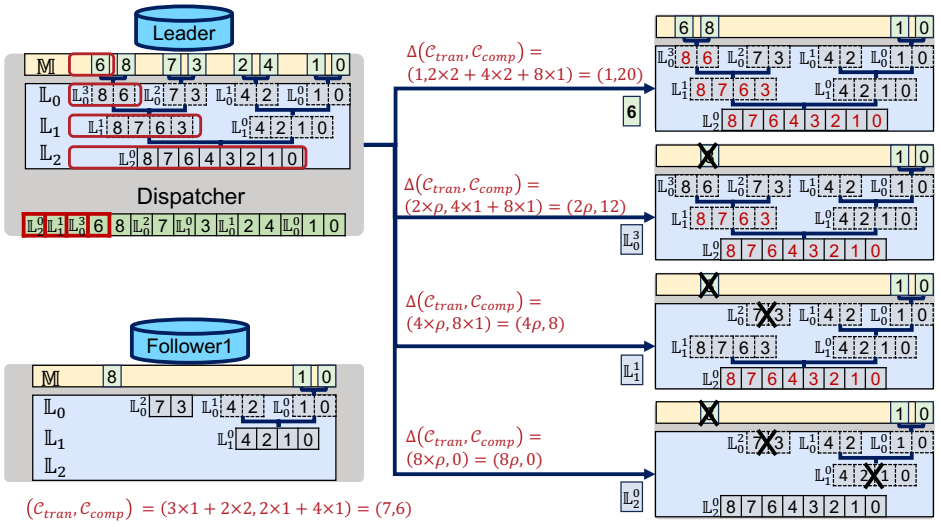


Fig. 4. Simple Example for Level Determination in LSM-Raft. The leader maintains a multi-level LSM-tree containing both atomic entries and compacted SSTables across levels $\mathbb{L}_0$, $\mathbb{L}_1$, and $\mathbb{L}_2$. When synchronizing with followers, the system determines the optimal SSTable transmission strategy by evaluating the transmission-compaction cost $\Delta(\mathrm{C_{tran}}, \mathrm{C_{comp}})$. Each arrow between the leader and follower indicates a potential transmission decision for a selected SSTable, along with its associated cost pair. For already cost, we have transmits about assuming the weights for compaction overhead per level are $\{1, 2, 4\}$ respectively. Follower 1 evaluates five possible options, each with a different combination of transmission and expected local compaction cost. The optimal combination $(3 \times 1 + 2 \times 2, 2 \times 1 + 4 \times 1)$ results in a total cost $(7, 6)$, indicating transmission size 7 and local compaction overhead 6 units. This selection minimizes overall effort compared to other alternatives such as $(8\rho, 0)$, which defers compaction entirely to the leader.

is the compression ratio), and a per-element transmission overhead δ. $\mathbf{L}_f^{ent}, \mathbf{L}_f^{sst}$ denotes the set of singleton entries and compacted SSTables.

$$C_{\text{trans}}(\mathbf{L}_f) = \sum_{E \in \mathbf{L}_f^{ent}} (|E| + \delta) + \sum_{E \in \mathbf{L}_f^{sst}} (\rho|E| + \delta) \tag{3}$$

$$= \sum_{E \in \mathbf{L}_f^{ent}} |E| + \rho \sum_{E \in \mathbf{L}_f^{sst}} |E| + |\mathbf{L}_f|\delta \tag{4}$$

Compaction cost captures the effort of incorporating logs into the local LSM-Tree. For each element E_i, the cost depends on the difference between the target LSM-Tree level and the original one.

$$C_{\text{comp}}(\mathbf{L}_f) = \sum_{E_i \in \mathbf{L}_f} \left(\text{Level}(\Phi(\mathbf{L}_f, \mathbf{S}_f)) - \text{Level}(E_i)\right) \cdot |E_i| \tag{5}$$

where the Level denotes the corresponding LSM-Tree level of the given states or elements. This reflects that compaction costs are higher when moving from lower to higher levels.

Overall cost is the sum of both costs for simplicity, $C^*(\mathbf{L}_f) = C_{\text{trans}}(\mathbf{L}_f) + C_{\text{comp}}(\mathbf{L}_f)$. This formula provides a unified metric to guide replication decisions.

Example 7. *Following the Example 6, we assume $\rho = 0.5$, $\delta = 10$, and $|E| = 1000$. For $\mathbf{L}_{v20}$ with 10 raw log elements:*

$$C_{trans}(\mathbf{L}_{v20}) = 10 \times 1000 + 10 \times 10 = 10100$$
$$C_{comp}(\mathbf{L}_{v20}) = 10 \times (1 - 0) \times 1000 = 10000$$

For $\langle E_{sst} \rangle$ of size 10000:

$$C_{trans}(\langle E_{sst} \rangle) = 0.5 \times 10000 + 10 = 5010$$
$$C_{comp}(\langle E_{sst} \rangle) = 0$$

To formalize this with weight preference $\alpha = 1, beta = 0.1$

$$C^*(\mathbf{L}_{v20}) = 10100 + 0.1 \times 10000 = 11100$$
$$C^*(\langle E_{sst} \rangle) = 5010$$

The overall cost comparison clearly favors E_{sst}. This demonstrates that compaction-aware, compression-optimized SSTable transfer achieves significantly lower overhead.

Compared with Example 4 and 5, Exmample 7 shows that when the index range is enlarged to 10,000, the overall cost is also reduced significantly (from 11,100 to 5,010), indicating improvement.

5.3 Optimization Objective and Strategy

Given a follower's current log $\mathbf{L}_f$ and a newly flushed SSTable E^{sst}, we define a candidate update $\mathbf{L}'_f = \mathbf{L}_f \cup \langle E^{\text{sst}} \rangle$. To select the best option, we consider all semantically equivalent log forms $\mathcal{E}_{\mathbf{L}_f}$ and select:

$$\mathbf{L}^* = \arg\min_{\mathbf{L} \in \mathcal{E}_{\mathbf{L}_f}} C^*(\mathbf{L}) \tag{6}$$

Example 8. *Follower 4 is at index 18, last received* $[15001, 16000]$. *A new SSTable covering* $[16001, 20000]$ *is available, then the log* $\mathbf{L}_{f_4}^{v20} = \langle [17001, 18000], [18001, 19000], [19001, 20000] \rangle$. *Assume $|E| = 1000$, $\rho = 0.5$, $\delta = 10$, all entries at level 0. Original cost and SSTable cost is as shown.*

$$C_{trans}(\mathbf{L}_{f_4}^{v20}) = 3030, \qquad C_{trans}(\langle E_{sst} \rangle) = 5010$$
$$C_{comp}(\mathbf{L}_{f_4}^{v20}) = 10000, \qquad C_{comp}(\langle E_{sst} \rangle) = 0$$
$$C^*(\mathbf{L}_{f_4}^{v20}) = 4030, \qquad C^*(\langle E_{sst} \rangle) = 5010$$

Hence, the original log yields lower total cost and should be used.

This strategy enables intelligent decisions when transmitting SSTables, improving replication throughput without sacrificing consistency. It is particularly beneficial for time-series or high-frequency ingest workloads.

5.4 SSTable Replacement Decision Algorithm

To efficiently decide whether to replace a series of raw log entries with a compressed SSTable, we introduce a lightweight, cost-delta-based decision algorithm. Instead of simulating the entire state machine, this method compares only the total transmission and compaction costs between the original log and the candidate SSTable. The principle is straightforward: If the cost of sending the SSTable is lower than that of sending and compacting the raw log entries it overlaps with, the SSTable should be chosen.

5.4.1 Algorithm Complexity. Let $\mathbf{L}_f$ be the current follower's pending log entries, and E_{sst} a newly flushed SSTable overlapping part of $\mathbf{L}_f$. r_{log} r_{sst}, o ,s are defined as shown in the Algorithm 2. The cost differences are computed as Lines 5 and 6 indicate.

If $\Delta C < 0$, then transmitting E_{sst} is considered more efficient as Line 8 shows. Otherwise, the original raw entries are chosen. This evaluation method is both fast and effective. It relies solely on metadata like timestamp ranges and record counts, and avoids heavyweight computation such as full LSM compaction simulation. Algorithm 2 involves only constant-time operations, including calculating ranges, overlaps, and costs. Both time and space complexity are $O(1)$, making the algorithm highly suitable for real-time usage in replication-intensive environments.

Algorithm 2 SSTableDecision

Require: Log sequence $\mathbf{L}_f$, SSTable E_{sst}, compression ratio ρ, transmission overhead δ
Ensure: `True` if SSTable is preferable, otherwise `False`

1: $r_{\text{log}} \leftarrow$ timestamp range of $\mathbf{L}_f$
2: $r_{\text{sst}} \leftarrow$ timestamp range of E_{sst}
3: $o \leftarrow$ overlap length of r_{log} and r_{sst}
4: $s \leftarrow |E_{\text{sst}}|$ ▷ Number of records in SSTable
5: $\Delta C_{\text{trans}} \leftarrow o - \rho \cdot s$
6: $\Delta C_{\text{comp}} \leftarrow -s$
7: $\Delta C \leftarrow \Delta C_{\text{trans}} + \Delta C_{\text{comp}}$
8: **if** $\Delta C < 0$ **then**
9: **return** `True`
10: **else**
11: **return** `False`
12: **end if**

5.4.2 Query Correctness. We explain that the query of the client's requests are still correct by skipping low level SSTable after applying Algorithm 2.

Consistency Despite Missing Lower-Level SSTables: As shown in Figure 4, missing a lower-level SSTable does not affect consistency. If a follower receives L_{i+1} but not L_i, three cases are possible: (A) L_{i+1} is generated from L_i and others—equivalence holds; (B) L_{i+1} is newer—L_i's data is preserved elsewhere; (C) L_{i+1} is older—L_i will be transmitted later. In all cases, the follower retains a consistent view of the committed data.

LSM-Raft adopts the standard Raft leader-based read model to ensure query correctness. As long as a majority of replicas have committed the log, client queries—served by the leader—are guaranteed to return the latest committed data. Thus, query results remain correct and unaffected by the state of lower-level SSTables.

6 IMPLEMENTATION

This section presents the system implementation of LSM-Raft, focusing on how the design principles are realized across the WAL, consensus, and storage layers. The architecture integrates the LSM-tree compaction model into a Raft-style consensus protocol while ensuring log consistency and fault tolerance.

6.1 System Overview

Figure 5 illustrates the architecture of LSM-Raft. The system is composed of three primary layers: **WAL Layer** is Responsible for capturing incoming write requests from clients and ensuring

persistence before replication. **Consensus Layer** handles log agreement across nodes and decides whether to replicate entries or compressed SSTables. **Storage Layer** maintains an LSM-tree-based storage engine and supports compaction and SSTable merging. We integrate LSM-Raft into two open-source database systems —TiDB [22] and Apache IoTDB [2, 41] —across the WAL, consensus, and storage.

6.2 Consensus Layer: SSTable Decision

A dedicated `Log Dispatcher` is responsible for feeding newly arrived log entries into the consensus layer. Depending on system state and follower lag, the dispatcher may either pass raw entries or trigger compaction to form SSTables for transmission.

The consensus layer is enhanced with a policy module that decides, for each follower, whether to replicate raw entries or compacted SSTables. This decision is based on the Algorithm 2 with then maintained `nextIndex` and `matchIndex`. In TiDB, we modify the RaftStore module to support generalized log element management. In IoTDB, we extend its built-in consensus module (PipeConsensus) to support log element switching.

6.3 Storage Layer and LSM Integration

At both leader and follower nodes, the local storage engine is built upon an LSM-tree. Once an entry or SSTable is deemed committed, it is flushed into the local LSM-tree through standard compaction procedures.

Each SSTable includes metadata identifying its compaction level, log index, and term, which is used during replication to ensure correctness. The compaction process at the leader is also reused to generate transferable SSTables that help reduce synchronization overhead. TiDB (via TiKV) uses RocksDB as its LSM-based storage engine, while IoTDB adopts an LSM-like storage engine based on the TsFile format. We adapt their compaction pipelines to support metadata-aware SSTable replay and verification.

6.4 Failure Recovery and Log Convergence

When a follower node crashes and later recovers, it resumes from its latest durable WAL state. The leader uses the `nextIndex` pointer to retry replication. Depending on the follower's current log span, the leader can choose to resend entries, trigger SSTable re-transmission, or compact overlapping ranges, as indicated in Algorithm 1.

Log convergence is ensured through the AppendEntries protocol: inconsistent logs are pruned and replaced based on log span conflict detection. The monotonicity of `matchIndex` and quorum-based commit logic guarantees safety despite compaction and transmission optimization. Failure recovery in TiDB and IoTDB reuses the native WAL replay mechanisms, with additional logic for element-level consistency checking.

7 EXPERIMENTS

This section evaluates the performance, efficiency, and fault tolerance of LSM-Raft through a series of experiments. We compare LSM-Raft against traditional Raft across various datasets and workload conditions to demonstrate its advantages in throughput and resource usage.

7.1 Setup

We evaluate two representative systems: TiDB [7, 22], designed for transactional and mixed OLTP/OLAP workloads, and Apache IoTDB [1, 41], targeting time-series ingestion workloads. Implementation and integration details of LSM-Raft within both systems are presented in Section 6.

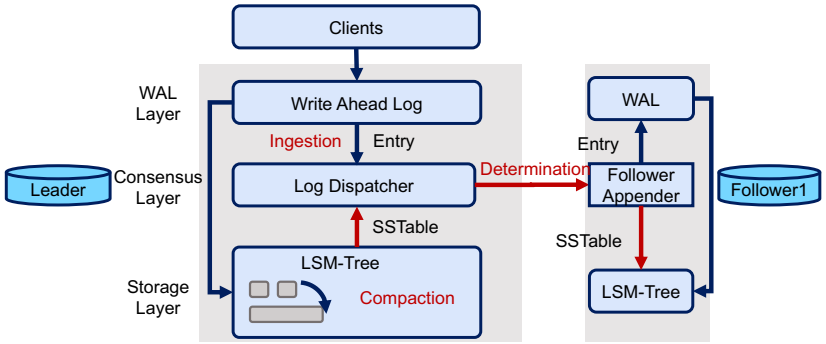


Fig. 5. The System Implementation of LSM-Raft

Unless otherwise specified, we use the default configuration settings (e.g., an SSTable size of 10,000) for each database.

7.1.1 Datasets. Four representative datasets, BoTIoT [26], EdgeIIoT [19], ToNIoT [10], CitiBike [17] are chosen for experiments. They are widely adopted for evaluating intrusion detection systems and distributed data analytics with an emphasis on their data sources, record scale, feature dimensions.

Table 2 summarizes key statistics of the above datasets.

Table 2. Datasets Statistics

Dataset	Year	Records	Features
BoTIoT [26]	2019	> 72M	Dozens
EdgeIIoT [19]	2022	~ 21M	61
ToNIoT [10]	2020	~30k/device	Varies
CitiBike [17]	2013-2025	> 300M	15+

7.1.2 Benchmark Workload. To demonstrate practical impact with realistic deployment scenarios from production, workloads are generated using two established benchmarking frameworks: YCSB [8] for TiDB and iot-benchmark [6] for IoTDB. For instance, Apache IoTDB is deployed in the urban rail vehicle monitoring system where each train is equipped with tens of thousands of sensors - such large-scale workloads can be accessed using iot-benchmark. Here, LSM-Raft's ability to cut overhead and adaptively reduce redundant data transmission can significantly improve system responsiveness (about 5%-9% in Figures 6) and resource efficiency (over 30% reduction in Figures 11, 12, 13).

To evaluate performance across a range of ingestion intensities, we vary the number of concurrent clients from 1 (light-ingestion) to 1024 (heavy-ingestion). Clients continuously issue append-heavy write operations targeted at the leader node, which replicates all writes synchronously to follower replicas.

7.1.3 Evaluation Metrics. We use several key metrics to assess system performance. **System throughput** (points/sec) measures the number of data points processed per second, reflecting the system's overall data ingestion capability. **CPU utilization** monitors the resource usage of each server node under varying workloads. **Compaction cost** records the number of data points involved in the compaction process, reflecting the overhead introduced by internal data merging.

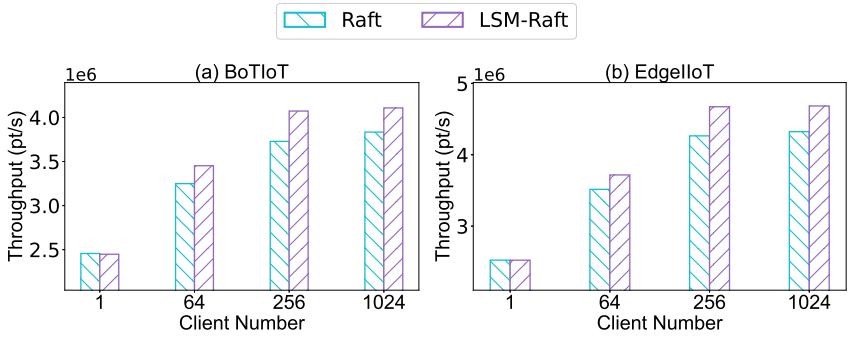


Fig. 6. System throughput in IoTDB.

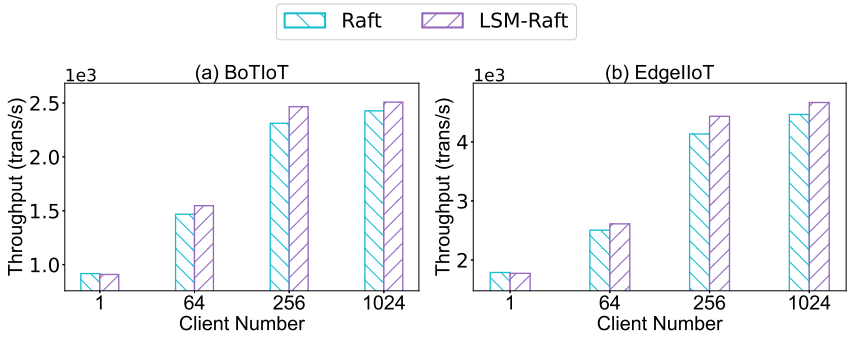


Fig. 7. System throughput in TiDB.

Network cost measures the bandwidth consumed during data replication and synchronization across servers, representing the communication overhead of the consensus protocol.

7.2 Overall Throughput

7.2.1 System Throughput. As described in Section 6, we implement LSM-Raft in both TiDB and Apache IoTDB. To validate its effectiveness across different database systems, we conduct experiments illustrated in Figures 6 and 7. Results show that LSM-Raft consistently improves system throughput about 3% to 10% in both TiDB and IoTDB.

7.2.2 High Ingestion. To further highlight the throughput gains, we set the client number range to [1, 1024]. As shown in Figures 6 and 7, when the number of clients is small (e.g., 1), the write pressure is low, and vanilla Raft can already achieve optimal performance. Hence, the throughput of Raft and LSM-Raft are nearly identical under light ingestion. As the number of clients increases, both systems demonstrate similar trends: with increasing concurrency, the throughput improves continuously, and the performance benefits of LSM-Raft become more pronounced. At 256 clients, throughput improvements reach 9.2% for IoTDB and 6.6% for TiDB. When client count reaches 1024, no further significant gains are observed, indicating that the system reaches an upper bound in I/O throughput, and additional clients no longer yield improvement.

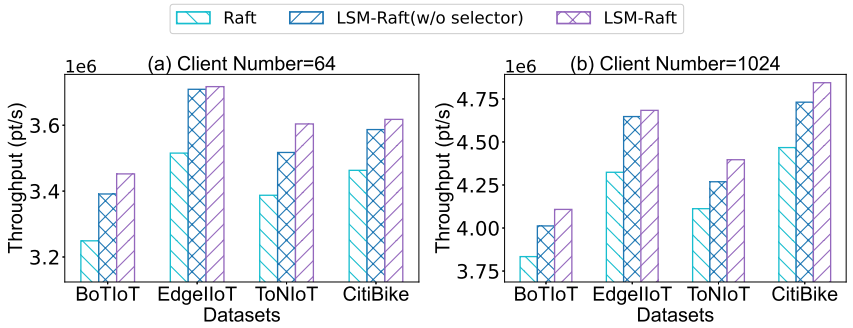


Fig. 8. System throughput wtih ablation.

Optimizing end-to-end throughput is inherently difficult in systems already optimized for high write performance. Throughput is influenced by factors such as flush intervals, memory management, and client-side buffering [34, 41], many of which lie beyond the scope of LSM-Raft. We argue that the observed 3-10% throughput improvement is meaningful in consensus-based systems, especially considering the difficulty of further optimizing already mature Raft implementations, while prior work such as NB-Raft reports comparable gains, e.g., 5% with 64 clients [24].

On one hand, as Figure 6 and 8 indicate, the higher the workload is, the more improvement LSM-Raft achieves in system through- put. On the other hand, LSM-Raft delivers more substantial gains in other resource dimensions. As shown in Figures 11, 12, and 13, our measurements show consistent reductions of over 30% in compaction, network and compcation cost.

7.2.3 Ablation Study. To evaluate the contributions of individual optimizations such as log generalization or the cost-based SSTable selector, we conduct an ablation study of component-wise comparisons. These include comparisons between: (1) vanilla Raft, (2) our LSM-Raft without cost-based SSTable selector, and (3) full LSM-Raft with both log generalization and cost-based SSTable selector. This breakdown clarifies the significance of both log generalization and cost-aware selector proposed in Section 3.1 and 5.4.

As shown in Figure 8, we examine performance across four datasets with varying client counts. We observe that the selector consistently yields additional throughput gains, typically between 1% and 3%, indicating its necessity even when raw SSTable transmission is already beneficial.

7.2.4 Parameter Tuning. In addition to evaluating client count and dataset type, we further investigate internal parameters such as flush frequency and compaction concurrency, both of which significantly influence performance.

Flush frequency controls how often data is flushed from memory to disk. Higher frequency (i.e., lower flush thresholds) results in smaller SSTables. As shown in Figure 9, when the flush threshold is set to 10,000 or 1,000, LSM-Raft yields 9% improvement over baseline Raft. However, when the threshold drops to 100, the generated SSTables are too small, and their transmission becomes as costly as raw entries. Conversely, at high thresholds (e.g., 100,000), many raw entries are sent before flushing occurs, undermining the benefits of SSTable-based transmission.

Compaction concurrency refers to the number of compaction threads running in parallel. At concurrency level 1, the system fails to keep up with the compaction load, degrading performance. Increasing concurrency to 16 allows the system to fully utilize its compaction capacity, leading to a 9% throughput gain with LSM-Raft. Beyond that, further increasing concurrency to 64 yields no noticeable benefit, indicating saturation of the compaction pipeline.

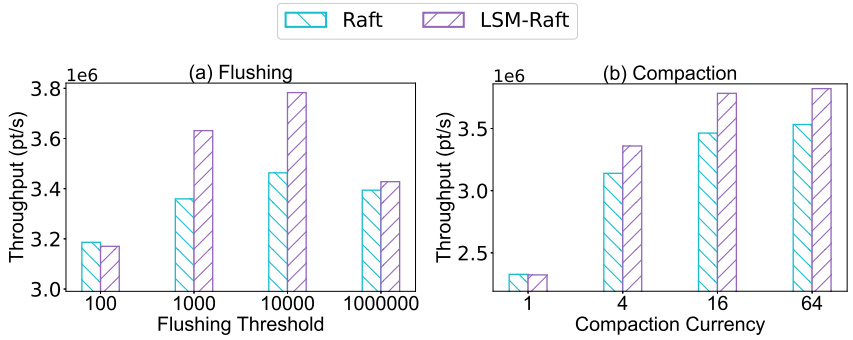


Fig. 9. System throughput with parameter tuning.

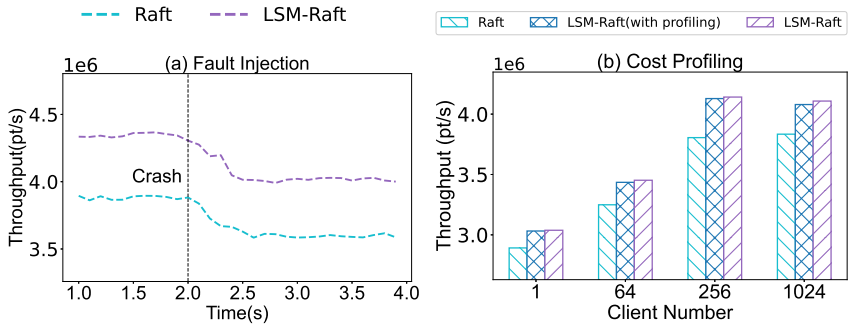


Fig. 10. System throughput with fault injection and cost profiling.

7.2.5 Fault Injection. To evaluate fault tolerance, we simulate a replica crash at 2s and monitor throughput recovery as shown in Figure 10(a). Both Raft and LSM-Raft exhibit an immediate drop, followed by rapid stabilization within 0.5 seconds, confirming their fault-tolerant capabilities. Notably, LSM-Raft maintains higher throughput throughout the experiment. Before the crash, it achieves 4.3M pt/s versus Raft's 3.9M pt/s; after recovery, it stabilizes at 4.0M pt/s while Raft remains at 3.6M pt/s. This demonstrates that LSM-Raft maintains fault-tolerant properties similar to Raft, quickly recovering to a stable throughput after the failure while still outperforming vanilla Raft.

7.2.6 Cost Profiling. We incorporate explicit runtime profiling of I/O latency and network bandwidth into our design. This enables feedback-driven adaptation of the synchronization strategy. Note that such profiling may incur additional overhead. The results in Figure 10(b) show that real-time cost profiling provides no significant improvement over log dispatcher heuristics. This suggests that follower log advancement implicitly reflects runtime conditions, supporting our choice of using a lower-overhead, dispatcher-based estimation strategy for dynamic adaptation.

7.3 Resource Utilization

To demonstrate LSM-Raft's improvements in resource utilization, we deploy a physical Raft cluster with 5 heterogeneous nodes, designed to emulate heterogeneous deployment scenarios. The heterogeneous hardware setup is characterized by two key aspects First, in terms of CPU resources,

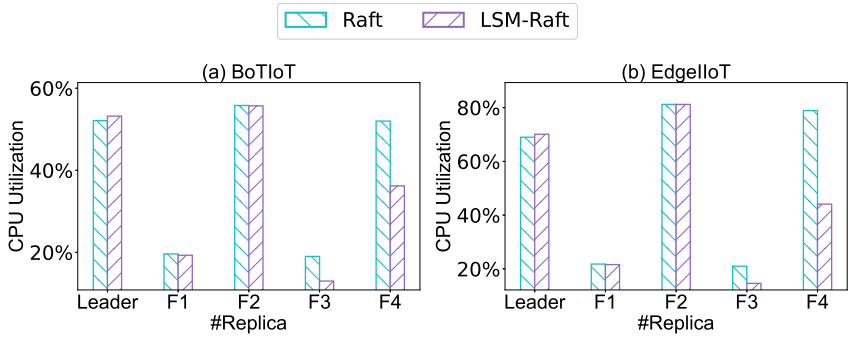


Fig. 11. CPU utilization across replicas.

replicas F1 and F3 (where F1 denotes Follower 1, etc.) are provisioned with 8-core Intel Xeon CPUs, whereas F2 and F4 are limited to 2-core CPUs. Second, in terms of network latency, F1 and F2 experience lower transmission delays than F3 and F4.

7.3.1 CPU Utilization. Figure 11 reports the CPU utilization of each node (leader and four followers) under a high-ingestion workload(client number equals 256) for both BoTIoT and EdgeIIoT datasets. For the BoTIoT dataset, LSM-Raft reduces the average CPU usage across the cluster from 40% to 35%, a relative reduction of approximately 11%. Notably, the follower replicas benefit the most: F3 from 19% to 13%, and F4 from 52% to 36%. The leader's CPU load increase slightly approximately 1%, as it is responsible for SSTable decision. The savings are more pronounced in the EdgeIIoT dataset, where follower nodes show reductions of up to 30%. This result reinforces that the performance advantages of LSM-Raft are not solely due to better throughput but are also backed by improved resource efficiency at the server level.

7.3.2 Network cost. Network overhead is another critical factor influencing replication efficiency and overall system scalability. Figure 12 presents the per-replica network transmission cost for both Raft and LSM-Raft under the BoTIoT and EdgeIoT. For BoTIoT, Leader's network usage drops from 440 MB to 304 MB (a 31% reduction), while F3/F4 achieves an even more pronounced reduction, from 110 MB to 59/23 MB (a 46%/79% reduction). This improvement is attributed to the protocol's ability to transmit pre-compacted SSTables, which encapsulate multiple log entries into a single file. This reduces the number of AppendEntries RPCs and avoids redundant transfer of overlapping entries.

7.3.3 Compaction cost. Compaction overhead directly reflects the system's internal I/O pressure and CPU usage during log maintenance. Figure 13 shows the total number of data points compacted by each replica under the BoTIoT dataset. Across all replicas, LSM-Raft significantly reduces compaction cost. For BoTIoT, the number of compacted points in F3 drops from 351 million to 247 million, representing a 29% reduction. F4 sees a larger reduction of 84.3% due to the transmission of higher level SSTables These gains are a direct result of LSM-Raft's design: by transmitting pre-compacted SSTables from the leader, follower nodes can avoid redundant local compaction. Together with reduced CPU and network cost (Figures 11 and 12), the results show that our proposed SSTable replication adaptively replaces the raw entry transmission for lower cost in a follower with limited resources. It is exactly one of the motivations in this study, cooperating LSM with Raft.

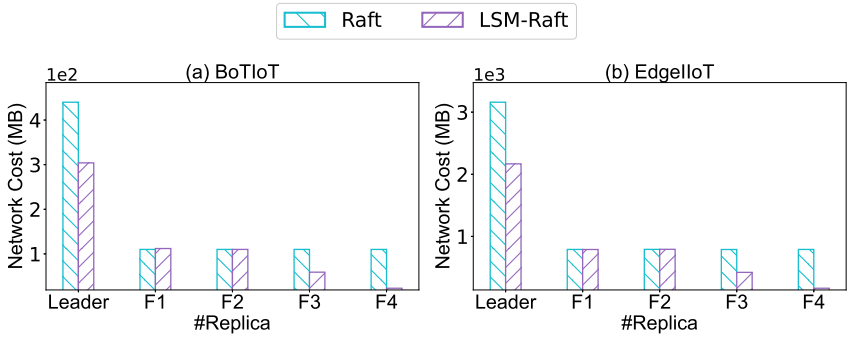


Fig. 12. Network cost across replicas.

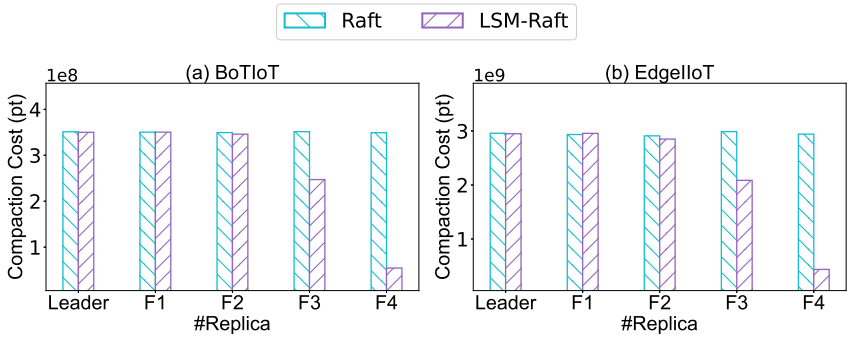


Fig. 13. Compaction cost across replicas

8 RELATED WORK

We review related work from two perspectives: Raft-based consensus protocols and LSM-tree storage engines.

8.1 Raft Consensus

Unlike Byzantine Fault Tolerant (BFT) consensus protocols [11, 12, 18], which are designed to tolerate arbitrary and potentially malicious behavior in adversarial environments, Crash Fault Tolerant (CFT) protocols such as Paxos [27, 28] and Raft [35, 36] focus on ensuring strong consistency under fail-stop failures while maintaining high availability and low operational complexity.

In distributed systems, ensuring strong consistency while maintaining low latency has long been a central challenge. To address this, many real-world systems adopt partial quorum strategies and accept eventual consistency as a trade-off. Standard Raft requires the leader to wait for a quorum to acknowledge each log entry before committing, which limits parallelism and throughput in high-ingestion scenarios. To mitigate this, Non-Blocking Raft [24] decouples log persistence from log commitment and allows ingestion to proceed before replication is complete. By tolerating limited inconsistency, it significantly improves throughput in IoT workloads with many sensors. Still, this approach introduces possible data loss and undermines Raft's original safety guarantees. Other work has focused on out-of-order execution to improve consensus efficiency. For instance, PolarFS [14] implements ParallelRaft to allow state machines to apply log entries out of order. However, this

design may encounter the "ghost entry" problem. These explorations reveal limitations in traditional Raft's linear log assumption, particularly for replication where logical equivalence and compacted state should be leveraged as first-class constructs.

8.2 LSM Storage Engines

The Log-Structured Merge-tree (LSM-tree) [32, 34] is the backbone of modern write-optimized databases, including time-series and key-value stores. It organizes data as a series of immutable sorted string tables (SSTables), which are periodically compacted to reduce storage redundancy and query overhead.

While LSM-trees excel at write throughput, they introduce unique challenges in distributed environments. Since compactions are executed independently across replicas, the same logical dataset may be encoded as different SSTables. This lack of byte-wise equivalence complicates log replication and increases recovery overhead. Raft's strict log matching requirements fail to exploit the semantic equivalence between compacted states, causing redundant computation and I/O.

LSM-trees require careful coordination of flush and compaction operations to maintain write efficiency under such conditions like Apache IoTDB [25, 47]. L-Store [37] is a unified engine for transactional and analytical processing. However, it does not use distributed consensus protocols like Raft, as it is designed for single-node setups. Other recent works such as Diff-Index [40] explore index structures that adapt to LSM-based update patterns, while systems like CRaft [43] and ECRaft [45] apply erasure coding to reduce replication cost. Yet none fully address the opportunity to generalize Raft's log model to accommodate the compaction semantics of LSM.

In this work, we extend these efforts by proposing a novel Raft variant that unifies compaction-aware storage and consensus replication. Our LSM-Raft protocol integrates log equivalence into the consensus model, enabling efficient transmission of SSTables while maintaining safety guarantees.

9 CONCLUSION

In this paper, we present **LSM Raft**, a generalized consensus protocol that adapts the traditional Raft algorithm to better align with the storage characteristics of Log Structured Merge (LSM) trees. By redefining log entries as generalized elements including compacted SSTables, and modeling the state machine as an LSM compaction function, LSM Raft enables more efficient replication without compromising consistency or fault tolerance.

We formally generalize Raft safety properties under the new log semantics, and verify them using TLA+, ensuring correctness even in the presence of out-of-order arrivals and overlapping SSTables. Furthermore, we propose a cost-aware log synchronization strategy that dynamically selects the optimal form of data transmission based on equivalence and replication cost.

Our implementation in Apache IoTDB demonstrates that LSM Raft can improve overall throughput by more than 20% and significantly reduce system overhead in real-world time series workloads. We believe LSM Raft bridges the semantic gap between log-based consensus and compaction-based storage engines, offering a promising direction for future consensus systems in high-ingest, storage-efficient environments.

ACKNOWLEDGMENTS

This work is supported in part by the National Natural Science Foundation of China (62021002, 92267203, 62232005), National Key Research & Development Plan (2025ZD1601701, 2024YFB3311901, 2021YFB3300500), Beijing National Research Center for Information Science and Technology (BNR2025RC01011), and Beijing Key Laboratory of Industrial Big Data System and Application. Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

REFERENCES

[1] 2025. Apache IoTDB. https://iotdb.apache.org.
[2] 2025. Code. https://github.com/LittleHealth/iotdb-raft
[3] 2025. Facebook RocksDB. https://github.com/facebook/rocksdb
[4] 2025. Google LevelDB. https://github.com/google/leveldb
[5] 2025. InfluxDB. https://www.influxdata.com/
[6] 2025. iot-benchmark. https://github.com/thulab/iot-benchmark
[7] 2025. PingCap TiKV. https://github.com/tikv/tikv
[8] 2025. PingCap YCSB. https://github.com/pingcap/go-ycsb
[9] 2025. TLA+ Proof. https://github.com/LittleHealth/iotdb-raft-tlaplus
[10] Abdullah Alsaedi, Nour Moustafa, Zahir Tari, Abdun Naser Mahmood, and Adnan Anwar. 2020. TON_IoT Telemetry Dataset: A New Generation Dataset of IoT and IIoT for Data-Driven Intrusion Detection Systems. *IEEE Access* 8 (2020), 165130–165150. https://doi.org/10.1109/ACCESS.2020.3022862
[11] Mohammad Javad Amiri, Chenyuan Wu, Divyakant Agrawal, Amr El Abbadi, Boon Thau Loo, and Mohammad Sadoghi. 2024. The Bedrock of Byzantine Fault Tolerance: A Unified Platform for BFT Protocols Analysis, Implementation, and Experimentation. In *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*. 371–400. https://www.usenix.org/conference/nsdi24/presentation/amiri
[12] Michael Barborak, Anton Dahbura, and Miroslaw Malek. 1993. The Consensus Problem in Fault-Tolerant Computing. *ACM Comput. Surv.* 25, 2 (1993), 171–220. https://doi.org/10.1145/152610.152612
[13] Huijie Cao, Chenfeng Huang, Shengchi Liu, Huiqi Hu, Minghao Zhao, Xuan Zhou, Yaofeng Tu, and Weining Qian. 2025. Aion: Live Migration for In-Memory Databases with Zero Downtime and Reduced Redundant Data Transfer. *Data Sci. Eng.* 10, 2 (2025), 212–229. https://doi.org/10.1007/S41019-024-00276-5
[14] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: An Ultra-low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *Proc. VLDB Endow.* 11, 12 (2018), 1849–1862. https://doi.org/10.14778/3229863.3229872
[15] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2 (2008), 4:1–4:26. https://doi.org/10.1145/1365815.1365816
[16] Giacomo Chiarot and Claudio Silvestri. 2023. Time Series Compression Survey. *ACM Comput. Surv.* 55, 10 (2023), 198:1–198:32. https://doi.org/10.1145/3560814
[17] CitiBike. 2025. *Citi Bike Histories*. https://ride.citibikenyc.com/system-data
[18] Allen Clement, Edmund L. Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. 2009. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*. 153–168. http://www.usenix.org/events/nsdi09/tech/full_papers/clement/clement.pdf
[19] Mohamed Amine Ferrag, Othmane Friha, Djallel Hamouda, Leandros Maglaras, and Helge Janicke. 2022. Edge-IIoTset: A New Comprehensive Realistic Cyber Security Dataset of IoT and IIoT Applications for Centralized and Federated Learning. *IEEE Access* 10 (2022), 40281–40306. https://doi.org/10.1109/ACCESS.2022.3165809
[20] Hua Guo, Yunhong Ji, and Xuan Zhou. 2024. The Development of a TLA$^+$ Verified Correctness Raft Consensus Protocol. In *Web and Big Data - 8th International Joint Conference, APWeb-WAIM 2024, Jinhua, China, August 30 - September 1, 2024, Proceedings, Part V*. 459–469. https://doi.org/10.1007/978-981-97-7244-5_40
[21] Fusheng Han, Hao Liu, Bin Chen, Debin Jia, Jianfeng Zhou, Xuwang Teng, Chuanhui Yang, Huafeng Xi, Wei Tian, Shuning Tao, Sen Wang, Quanqing Xu, and Zhenkun Yang. 2024. PALF: Replicated Write-ahead Logging for Distributed Databases. *Proc. VLDB Endow.* 17, 12 (2024), 3745–3758. https://doi.org/10.14778/3685800.3685803
[22] 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. 2020. TiDB: A Raft-based HTAP Database. *Proc. VLDB Endow.* 13, 12 (2020), 3072–3084. https://doi.org/10.14778/3415478.3415535
[23] Mingtao Ji, Yibo Jin, Zhuzhong Qian, Tuo Cao, and Bao-Liu Ye. 2025. Orchestrating In-Network Aggregation for Distributed Machine Learning via In-Band Network Telemetry. *J. Comput. Sci. Technol.* 40, 1 (2025), 196–214. https://doi.org/10.1007/S11390-024-3342-Y
[24] Tian Jiang, Xiangdong Huang, Shaoxu Song, Chen Wang, Jianmin Wang, Ruibo Li, and Jincheng Sun. 2023. Non-Blocking Raft for High Throughput IoT Data. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. 1140–1152. https://doi.org/10.1109/ICDE55515.2023.00092
[25] Yuyuan Kang, Xiangdong Huang, Shaoxu Song, Lingzhe Zhang, Jialin Qiao, Chen Wang, Jianmin Wang, and Julian Feinauer. 2022. Separation or Not: On Handing Out-of-Order Time-Series Data in Leveled LSM-Tree. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. 3340–3352. https:

//doi.org/10.1109/ICDE53745.2022.00315

[26] Nickolaos Koroniotis, Nour Moustafa, Elena Sitnikova, and Benjamin P. Turnbull. 2019. Towards the development of realistic botnet dataset in the Internet of Things for network forensic analytics: Bot-IoT dataset. *Future Gener. Comput. Syst.* 100 (2019), 779–796. https://doi.org/10.1016/j.future.2019.05.041

[27] Leslie Lamport. 2002. Paxos Made Simple, Fast, and Byzantine. In *Procedings of the 6th International Conference on Principles of Distributed Systems. OPODIS 2002, Reims, France, December 11-13, 2002.* 7–9.

[28] Leslie Lamport. 2006. Fast Paxos. *Distributed Comput.* 19, 2 (2006), 79–103. https://doi.org/10.1007/S00446-006-0005-X

[29] Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, and Yu Zheng. 2023. Elf: Erasing-based Lossless Floating-Point Compression. *Proc. VLDB Endow.* 16, 7 (2023), 1763–1776. https://doi.org/10.14778/3587136.3587149

[30] Yuetai Li, Yixuan Fan, Lei Zhang, and Jon Crowcroft. 2023. RAFT Consensus Reliability in Wireless Networks: Probabilistic Analysis. *IEEE Internet Things J.* 10, 14 (2023), 12839–12853. https://doi.org/10.1109/JIOT.2023.3257402

[31] Yanhong A. Liu, Saksham Chand, and Scott D. Stoller. 2019. Moderately Complex Paxos Made Simple: High-Level Executable Specification of Distributed Algorithms. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019.* 15:1–15:15. https://doi.org/10.1145/3354166.3354180

[32] Chen Luo and Michael J. Carey. 2020. LSM-based storage techniques: a survey. *VLDB J.* 29, 1 (2020), 393–418. https://doi.org/10.1007/S00778-019-00555-Y

[33] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is more consensus in Egalitarian parliaments. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013.* 358–372. https://doi.org/10.1145/2517349.2517350

[34] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385. https://doi.org/10.1007/S002360050048

[35] Diego Ongaro. 2014. *Consensus: bridging theory and practice.* Ph. D. Dissertation. Stanford University, USA. https://searchworks.stanford.edu/view/10608105

[36] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC 2014, Philadelphia, PA, USA, June 19-20, 2014.* 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[37] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. 2018. L-Store: A Real-time OLTP and OLAP System. In *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018.* 540–551. https://doi.org/10.5441/002/EDBT.2018.65

[38] Ermin Sakic and Wolfgang Kellerer. 2018. Response Time and Availability Study of RAFT Consensus in Distributed SDN Control Plane. *IEEE Trans. Netw. Serv. Manag.* 15, 1 (2018), 304–318. https://doi.org/10.1109/TNSM.2017.2775061

[39] Subhadeep Sarkar, Kaijie Chen, Zichen Zhu, and Manos Athanassoulis. 2022. Compactionary: A Dictionary for LSM Compactions. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022.* 2429–2432. https://doi.org/10.1145/3514221.3520169

[40] Wei Tan, Sandeep Tata, Yuzhe Richard Tang, and Liana L. Fong. 2014. Diff-Index: Differentiated Index in Distributed Log-Structured Data Stores. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014.* 700–711. https://doi.org/10.5441/002/EDBT.2014.76

[41] Chen Wang, Jialin Qiao, Xiangdong Huang, Shaoxu Song, Haonan Hou, Tian Jiang, Lei Rui, Jianmin Wang, and Jiaguang Sun. 2023. Apache IoTDB: A Time Series Database for IoT Applications. *Proc. ACM Manag. Data* 1, 2 (2023), 195:1–195:27. https://doi.org/10.1145/3589775

[42] Rihong Wang, Lifeng Zhang, Quanqing Xu, and Hang Zhou. 2019. K-Bucket Based Raft-Like Consensus Algorithm for Permissioned Blockchain. In *25th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2019, Tianjin, China, December 4-6, 2019.* 996–999. https://doi.org/10.1109/ICPADS47876.2019.00152

[43] Zizhong Wang, Tongliang Li, Haixia Wang, Airan Shao, Yunren Bai, Shangming Cai, Zihan Xu, and Dongsheng Wang. 2020. CRaft: An Erasure-coding-supported Version of Raft for Reducing Storage Cost and Network Cost. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020.* 297–308. https://www.usenix.org/conference/fast20/presentation/wang-zizhong

[44] Jinzhao Xiao, Yuxiang Huang, Changyu Hu, Shaoxu Song, Xiangdong Huang, and Jianmin Wang. 2022. Time Series Data Encoding for Efficient Storage: A Comparative Analysis in Apache IoTDB. *Proc. VLDB Endow.* 15, 10 (2022), 2148–2160. https://doi.org/10.14778/3547305.3547319

[45] Mingwei Xu, Yu Zhou, Yuanyuan Qiao, Kai Xu, Yu Wang, and Jie Yang. 2021. ECRaft: A Raft Based Consensus Protocol for Highly Available and Reliable Erasure-Coded Storage Systems. In *27th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2021, Beijing, China, December 14-16, 2021.* 707–714. https://doi.org/10.1109/ICPADS53394.2021.00094

[46] Xiaofeng Yu, Ruyan Liu, Lewis Nkenyereye, Zhiming Wang, and Yongjun Ren. 2024. ACRS-Raft: A Raft Consensus Protocol for Adaptive Data Maintenance in the Metaverse Based on Cauchy Reed-Solomon Codes. *IEEE Trans. Consumer*

Electron. 70, 1 (2024), 3792–3801. https://doi.org/10.1109/TCE.2024.3373435

[47] Xiaojian Zhang, Hongyin Zhang, Shaoxu Song, Xiangdong Huang, Chen Wang, and Jianmin Wang. 2023. Backward-Sort for Time Series in Apache IoTDB. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. 3196–3208. https://doi.org/10.1109/ICDE55515.2023.00245

Received April 2025; revised July 2025; accepted August 2025