



How Much Can RocksDB Chew? Achieving Near-Zero Write Stalls with Sustainable RocksDB

Hojin Shin
Dankook University
Yongin, Republic of Korea
hojin03s@dankook.ac.kr

Seehwan Yoo*
Dankook University
Yongin, Republic of Korea
seehwan.yoo@dankook.ac.kr

Yongmin Lee
Dankook University
Yongin, Republic of Korea
ym.lee@dankook.ac.kr

Jongmoo Choi*
Dankook University
Yongin, Republic of Korea
choijm@dankook.ac.kr

ABSTRACT

Modern data-intensive applications, from microservices to real-time AI serving, demand consistently low tail latency from backend storage. However, log-structured merge-tree-based key-value stores like RocksDB are structurally prone to unpredictable *write stalls*. These stalls stem from a fundamental architectural decoupling of foreground write ingress and background data reorganization. By design, the system absorbs foreground writes at maximum speed without monitoring its actual time-varying compaction capacity. As a result, it accumulates internal pressure until rigid capacity thresholds are breached, triggering reactive safeguards that abruptly freeze all foreground writes. Relying on this reactive “stop-and-go” approach induces a persistent limit-cycle behavior, undermining long-run predictability and strict latency guarantees.

We reframe write stalls as a continuous control problem. S-RocksDB is a sustainable admission controller that regulates foreground ingress to match the system’s time-varying compaction capacity. Since this capacity varies at runtime, S-RocksDB employs online reinforcement learning to discover a sustainable admission rate. To ensure safe learning, a three-state operational model (SAFE, SEMI-SAFE, UNSAFE) confines exploration to stable conditions and deploys deterministic guardrails before stalls can occur. In 24-hour evaluations, S-RocksDB reduces over 64.3M stalled writes to just 69, bounds P99.99 tail latency to sub-0.11 ms, and delivers predictable throughput with the lowest resource footprint among all compared systems.

PVLDB Reference Format:

Hojin Shin, Yongmin Lee, Seehwan Yoo, and Jongmoo Choi. How Much Can RocksDB Chew? Achieving Near-Zero Write Stalls with Sustainable RocksDB. PVLDB, 19(11): 3202 – 3215, 2026.
doi:10.14778/3836663.3836683

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/shinhojin/Sustainable-RocksDB>.

*Corresponding authors

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836683

1 INTRODUCTION

In the era of modern cloud infrastructure, data-intensive workloads have imposed unprecedented demands on backend storage systems. Examples include large language model serving, personalized recommendations [27], real-time analytics, and microservices architectures [25]. These applications require not only high aggregate throughput but, more critically, strict guarantees on tail latency. Even millisecond-level delays at the 99th or 99.9th percentile can cascade through distributed call chains, severely degrading user experience and Service Level Agreements (SLAs) [21, 25]. As the foundational storage engine for many of these hyper-scale services, Log-Structured Merge-tree (LSM) [44]-based Key-Value Stores (KVSs) like RocksDB [24] face the heavy burden of providing consistently low latency under relentlessly high, time-varying write pressure.

However, satisfying this demand is structurally challenging due to the inherent decoupling of foreground write ingestion and background data reorganization (flush and compaction) in LSM-KVSs. When the incoming write rate persistently outpaces the system’s background processing capacity, internal pressure accumulates. This pressure manifests as proliferating Level-0 (L0) files and soaring pending compaction bytes, ultimately culminating in *write stalls*. To prevent unbounded resource exhaustion and out-of-memory crashes, reactive safeguards abruptly halt or throttle foreground traffic, trading latency consistency for temporary self-preservation. Consequently, this reactive paradigm induces a persistent limit-cycle behavior, shattering the predictable tail-latency required by modern real-time applications.

Historically, storage systems and research communities have attempted to mitigate these stall-induced oscillations through various architectural and tuning interventions. Earlier solutions like SILK [5] and MatrixKV [57] alleviate flush/compaction bottlenecks by introducing dedicated I/O schedulers or segregating NVM tiers. Similarly, automated tuning frameworks (e.g., Endure [30], CDB-Tune [60]) leverage machine learning to optimize internal configurations. More recently, state-of-the-art approaches (e.g., ADOC [58], RBC [55], RaKV [61]) dynamically adjust internal resources or utilize bandwidth-aware controllers to smooth internal dataflow.

Yet, these advancements predominantly focus on maximizing background capacity or managing I/O contention, rather than discovering a strictly sustainable admission limit. Consequently, they still rely on a fundamentally reactive, threshold-triggered binary

lens for control: utilizing static rate limiting or forcibly throttling incoming traffic only *after* safe operating margins vanish. This reactive paradigm fails to recognize the multi-dimensional, pre-stall accumulation of pressure. Essentially, treating a write stall as an emergency stop precludes modern KVSs from being sustainable and predictable. Instead, KVSs should navigate the continuous pressure dynamics to avoid stalls and guide it to a balanced, stable state.

In this paper, we present S-RocksDB (Sustainable RocksDB), a continuous, sustainable admission controller designed to dynamically discover and sustain the optimal ingestion rate. S-RocksDB achieves this by continuously regulating write admission so that admitted foreground ingress matches the system’s time-varying background processing capacity ($B_{\text{int}}(t)$), ensuring that ingestion never persistently exceeds what the system can digest.

Tracking a sustainable ingestion rate is inherently complex, as foreground writes, Level-0 growth, and compaction debt interact in nonlinear and multi-dimensional ways that render rigid heuristics and manual tuning ineffective. To address this challenge, S-RocksDB employs a lightweight reinforcement learning (Q-learning) [52, 56] controller that continuously adjusts the admission multiplier based on observed system state. To ensure stability during exploration, the Q-learning agent optimizes admission exclusively when the system is in a stable, stall-free state (SAFE), while deterministic guardrails at the SEMI-SAFE boundary preemptively absorb sudden pressure surges, preventing the learning process from escalating into catastrophic stalls. This encapsulation of learning within the SAFE region and deterministic guardrails at the SEMI-SAFE boundary enables sustainable admission control with negligible runtime overhead.

The main contributions of this paper are as follows:

- We systematically analyze the multi-dimensional causes of write stalls in RocksDB, demonstrating the limitations of conventional limit-cycle controls and motivating the need to track time-varying background capacity $B_{\text{int}}(t)$.
- We design a Q-learning-based admission controller that learns the nonlinear relationship between ingress traffic and compaction debt online, operating out-of-process at 1-second epochs with negligible overhead.
- We introduce a hierarchical safety model (SAFE, SEMI-SAFE, UNSAFE) that confines Q-learning optimization to the SAFE region while enforcing deterministic guardrails near critical boundaries.
- We conduct extensive 24-hour microbenchmarks and YCSB evaluations, showing near-elimination of stalled writes and sub-0.11 ms P99.99 write latency, whereas prior baselines exhibit millisecond-level tail spikes.

2 BACKGROUND

This section reviews RocksDB’s write path and its built-in stall safeguards. We focus on factual system behavior and terminology needed for our later formulation and control design.

2.1 Write Path in RocksDB

RocksDB is an LSM-KVS that strictly decouples foreground write ingestion from background data reorganization. As shown in Figure 1,

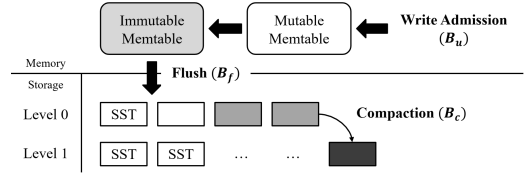


Figure 1: Write processing pipeline in RocksDB.

incoming writes are initially buffered in memory as memtables and subsequently persisted to storage via flush.

When a mutable memtable becomes full, it is converted to an immutable memtable, which is subsequently flushed to disk as a Sorted String Table (SST), initially placed in Level-0 (L0) [12]. Depending on the configuration, writes may also be logged to a write-ahead log (WAL). In this paper, however, our focus is strictly on stall behaviors primarily driven by flush and compaction backlogs, as well as their related pressure signals.

Compaction merges and reorganizes SSTs across levels (e.g., $L0 \rightarrow L1$ and below), discards obsolete data, and minimizes key range overlaps. Because compaction depends on the existing on-disk layout (e.g., overlap, level sizes, and scheduling), its effective processing rate inherently fluctuates over time. Crucially, this cascading reorganization creates a *fluid-like asynchronous delay* between foreground write throttling and actual background pressure relief. Even if incoming writes are abruptly halted, data already accumulated in memory buffers and L0 continues to cascade downward; therefore, internal pressure cannot be neutralized instantly. While this architecture affords high write throughput, it renders runtime stability highly sensitive to the balance between write ingress and background processing capacity.

2.2 Write Stall and Threshold-Based Safeguards

To prevent unbounded accumulation of internal backlog when background processing fails to keep pace with foreground writes, RocksDB incorporates threshold-based safeguard mechanisms. When internal pressure grows beyond configured thresholds, RocksDB exerts backpressure on the write path, delaying or temporarily blocking foreground writes.

Operationally, two stall-related interventions are commonly observed: *write slowdown*, where incoming writes are intentionally delayed to constrain the ingestion rate, and *write stop*, where writes are blocked until internal conditions recover. These protective actions are triggered and released based on strict threshold checks against internal pressure signals.

Typical signals related to stall safeguards include: (i) L0 pressure (e.g., too many L0 SST files), (ii) compaction backlog (e.g., pending compaction bytes indicating accumulated compaction debt), and (iii) memory/flush pressure (e.g., accumulation of immutable memtables or flush backlog risking memory exhaustion). A defining characteristic of these safeguards is their reactive, threshold-triggered nature: they activate only after specific metrics breach configured limits, and deactivate once those signals retreat below safe thresholds. Because these safeguards are purely threshold-triggered, they intervene only after internal pressure escalates, often resulting in oscillatory, stop-and-go behavior under continuous write workloads.

This limitation highlights the need for a proactive control mechanism that regulates ingestion smoothly before critical thresholds are breached.

2.3 Rate Limiting Mechanisms

Beyond reactive stall protections, RocksDB provides rate-limiting mechanisms to cap I/O throughput, thereby indirectly influencing the pace of background work and write ingestion. In practice, static rate caps are often employed as a workaround to reduce performance variability and lower the likelihood of triggering stall safeguards under continuous write-intensive workloads.

However, static rate limiting remains fundamentally a rigid configuration mechanism: it caps overall throughput but neither explicitly models the system’s operating states nor adapts to its time-varying background capacity. Consequently, the “optimal” limit is highly dependent on evolving workloads. A static limit may thus either underutilize available system resources or ultimately fail to prevent stalls as background performance drifts.

The next section empirically characterizes how write stalls arise in RocksDB and highlights limitations of purely threshold-triggered safeguards and static rate control.

3 MOTIVATIONS

Write stalls appear as threshold-triggered events, but they usually reflect accumulated imbalance between foreground write ingress and background flush/compaction capacity. This imbalance is due to the structural instability of LSM-KVSs: writes are admitted in the foreground, while data reorganization is deferred to background flush and compaction. Thus, reacting after a stall can temporarily relieve existing pressure, but it cannot prevent internal pressure from accumulating again under sustained writes. We therefore motivate S-RocksDB from two limitations of existing admission behavior: reactive threshold safeguards and static write-rate control.

3.1 Motivation 1: Structural Instability of Threshold-Triggered Safeguards

The write pipeline of an LSM-KVS can be viewed as a delayed flow system. Let $x_0(t)$ denote accumulated L0 pressure, $u(t)$ the foreground write ingress, and $c_0(t)$ the effective background compaction rate draining L0. The pressure dynamics can be expressed as:

$$\frac{dx_0(t)}{dt} = u(t) - c_0(t) \quad (1)$$

Because flush and compaction are asynchronous, $c_0(t)$ cannot react instantaneously to changes in $u(t)$. Thus, the system must continuously balance foreground input against a time-varying and delayed background service rate.

RocksDB’s built-in stall mechanisms regulate this imbalance through threshold-triggered safeguards. When accumulated pressure $x_0(t)$ crosses a configured limit x_{th} , foreground writes are slowed or stopped. Abstractly, this stop-and-go behavior resembles a multi-step *bang-bang* controller:

$$u(t) = \begin{cases} u_{max} & \text{if } x_0(t) < x_{th} \\ 0 & \text{if } x_0(t) \geq x_{th} \end{cases} \quad (2)$$

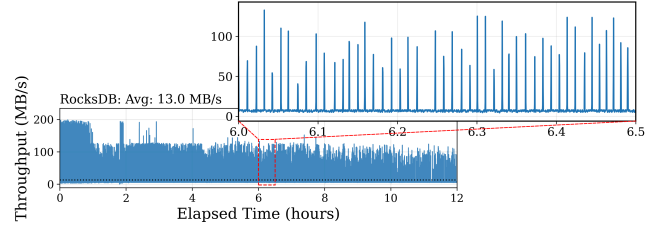


Figure 2: Throughput oscillation under RocksDB.

Although RocksDB also applies write slowdown before a hard stop, the control structure remains discontinuous: the system reacts after pressure approaches configured boundaries instead of explicitly tracking a stable operating point where $u(t) \approx c_0(t)$. In delayed systems, such discontinuous control is prone to limit-cycle behavior [4]: the system admits writes aggressively, overshoots because background work cannot drain pressure immediately, stalls to recover, and then re-enters aggressive admission.

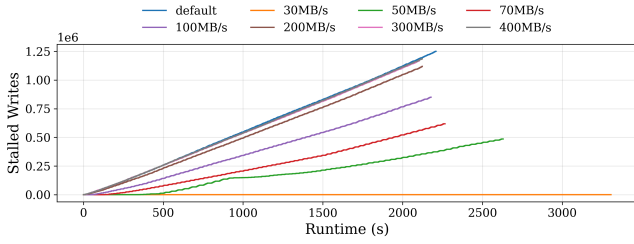
Figure 2 illustrates this behavior under a 12-hour *fillrandom* workload, zoomed into a representative 30-minute window starting at the 6-hour mark. RocksDB repeatedly alternates between high ingestion and reactive stall intervention, producing sharp throughput oscillations and 1.36M stalled writes in this window. This behavior is not only a matter of poorly chosen thresholds. LSM pressure is multi-dimensional: L0 files, compaction debt, and memtable/flush pressure evolve at different rates. Since RocksDB’s default safeguards react to threshold breaches of these signals, they do not explicitly model the continuous pre-stall region where pressure is already accumulating. Therefore, threshold tuning may change the frequency or amplitude of oscillations, but it cannot by itself provide smooth and proactive admission control under time-varying background capacity.

3.2 Motivation 2: Limitations of Static Write Rate Control

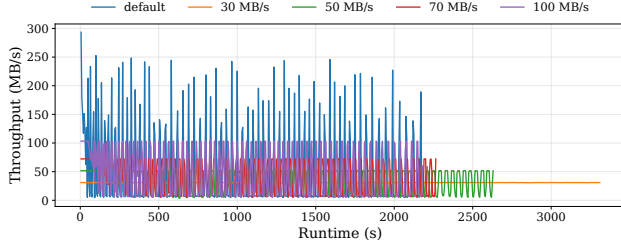
A common workaround for reducing stalls is to impose a fixed upper bound on the admitted write rate. To evaluate this approach, we ran a RocksDB *fillrandom* experiment inserting 100M key-value pairs under static write-rate limits from 400 MB/s down to 30 MB/s, and compared them with the default configuration without a static limit. We measure stall impact using cumulative stalled writes, i.e., the number of foreground writes forced to wait due to RocksDB’s stall mechanisms.

Figure 3a shows that static throttling reduces stalls only when the cap becomes highly conservative. Without a static limit, RocksDB accumulates 1.25M stalled writes. Moderate caps still leave substantial stall impact: even a 100 MB/s cap leaves 0.85M stalled writes, or 68.06% of the default. Only the most conservative 30 MB/s cap eliminates stalled writes within the observed window. However, Figure 3b shows that this comes by suppressing throughput peaks rather than by adapting to the actual time-varying background capacity. Moreover, a rate that is safe in one finite run may not remain safe as compaction dynamics, data distribution, workload mix, or device behavior change over time.

Static write-rate control therefore exposes a fundamental utilization and stability trade-off. Moderate throttling sacrifices peak



(a) Cumulative stalled writes under different static write rate limits



(b) Write throughput over time under different static write rate limits

Figure 3: Stalled writes and throughput under static rates.

throughput while still allowing many writes to be delayed by stalls, whereas eliminating stalls requires an overly conservative cap that underutilizes available capacity. These results motivate a dynamic admission mechanism that continuously adapts to internal pressure instead of committing to a fixed rate.

Takeaway. The two motivations point to the same conclusion. Reactive threshold safeguards are prone to stop-and-go oscillations in delayed LSM pipelines, while static rate limits cannot track runtime capacity drift. Write stall management should therefore be treated not as an emergency reaction after thresholds are crossed, but as a continuous admission-control problem that proactively tracks a sustainable ingestion rate.

4 SYSTEM DYNAMICS AND ADMISSION PROBLEM

Building upon the takeaways from Section 3, this section formalizes write admission as a dynamic tracking problem to stabilize internal pressure. Rather than treating a stall as an isolated event to detect and react to, we model it as the visible outcome of an accumulated imbalance between *foreground write ingress* and time-varying *background processing capacity* (flush and compaction). Our objective is to sustain stable long-run write admission while structurally breaking the limit cycles that cause stalls.

To reason about sustainability over long executions, we abstract the system using the bandwidth/pressure dynamics in Figure 4. This abstraction highlights where admission control intervenes (at the ingress) and why time-varying background service rates make static admission fragile.

4.1 Bandwidth Model

To begin with, we discretize time into control epochs. For each epoch t , we define:

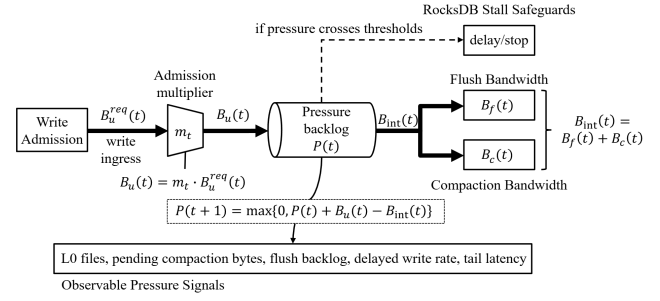


Figure 4: Bandwidth and pressure dynamics framework: admitted ingress (B_u) versus time-varying background capacity (B_{int}) and resulting pressure.

- $B_u^{req}(t)$: requested foreground write ingress,
- $B_u(t)$: admitted foreground write ingress,
- $B_f(t)$: effective flush bandwidth (memory $\rightarrow$ storage),
- $B_c(t)$: effective compaction bandwidth (on-disk reorganization).

We aggregate the instantaneous internal service capacity as:

$$B_{int}(t) = B_f(t) + B_c(t). \quad (3)$$

Building upon the continuous fluid system established in Equation 1, if admitted ingress persistently exceeds internal service, backlog accumulates. The discrete-time pressure dynamics model is expressed as:

$$P(t+1) = \max\{0, P(t) + B_u(t) - B_{int}(t)\}, \quad (4)$$

where $P(t)$ is an abstract internal pressure/backlog proxy (e.g., compaction debt). Importantly, $B_f(t)$ and $B_c(t)$ are highly *time-varying* due to evolving data distribution, level organization, compaction scheduling, and device behavior. This mathematical variability formalizes the inescapable empirical drift observed in Motivation 2, proving why a static admission rate B_u is fundamentally fragile and cannot guarantee long-run stability.

In S-RocksDB, we do not require direct estimation of $B_f(t)$ or $B_c(t)$. Instead, we treat runtime pressure signals as partial observations of accumulated imbalance; signals such as pending compaction bytes and L0 growth reflect the integral effect of $B_u(t) - B_{int}(t)$ and are thus suitable for continuous flow control.

4.2 Observable Pressure Signals and Stall Impact

Write stall manifests as foreground writes being delayed or blocked, but the underlying causes are heterogeneous. In practice, we summarize internal pressure using runtime-observable signals that are also closely related to RocksDB’s safeguards:

- L0 pressure (e.g., L0 file count),
- compaction pressure (e.g., pending compaction bytes),
- memtable/flush pressure (e.g., immutable memtable accumulation or flush backlog).

Although L0 file count is discrete, it serves as an excellent proxy for the accumulated data volume since individual SST files naturally fall within a bounded size range.

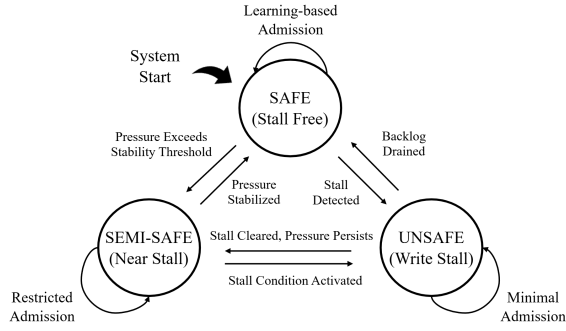


Figure 5: Three operational states in S-RocksDB.

For evaluation and controller-side stall detection, we distinguish between (i) whether stall protection is engaged and (ii) how much it affects foreground requests. We use *stalled writes* to quantify stall impact: the cumulative number of write operations that were forced to wait due to stall mechanisms (stop/delay).

4.3 Operational State Model of S-RocksDB

To explicitly represent pre-stall dynamics, S-RocksDB uses three operational states, $\sigma(t)$, in Figure 5. SAFE denotes stall-free execution with sufficiently low internal pressure, where admission can be cautiously increased. SEMI-SAFE denotes a near-stall state where stall is not yet observed but pressure approaches critical levels; admission should be conservative to avoid entering stall. UNSAFE denotes a stall-active condition (writes are delayed/blocked) or a short recovery window immediately after stall, where draining backlog is prioritized. The states are inferred by a controller from runtime signals and stall observations.

4.4 Admission Control Objective: Limit Cycle Breaking

S-RocksDB regulates write admission through a scalar multiplier m_t applied to requested ingress:

$$B_u(t) = m_t \cdot B_u^{req}(t). \quad (5)$$

The multiplier is constrained to a safe range:

$$m_t \in [m_{\min}, m_{\max}], \quad (6)$$

and is updated incrementally to avoid oscillation:

$$m_{t+1} = \text{clip}(m_t + \Delta m_t, m_{\min}, m_{\max}), \quad (7)$$

where $\text{clip}(x, a, b) = \max(a, \min(x, b))$.

The control objective is to break the destructive limit cycles proven in Motivation 1 and proactively converge to a continuous, sustainable operating equilibrium ($B_u(t) \approx B_{\text{int}}(t)$). Concretely, we aim to (i) proactively adapt to capacity drift over long runs rather than reacting to emergencies, (ii) maintain stable, high admission without large throughput oscillations, and (iii) minimize time spent in the UNSAFE recovery state.

To achieve this convergence safely, Section 5 introduces a proactive control architecture that infers the operational state $\sigma(t)$ and applies state-dependent admission policies, including safe-region reinforcement learning.

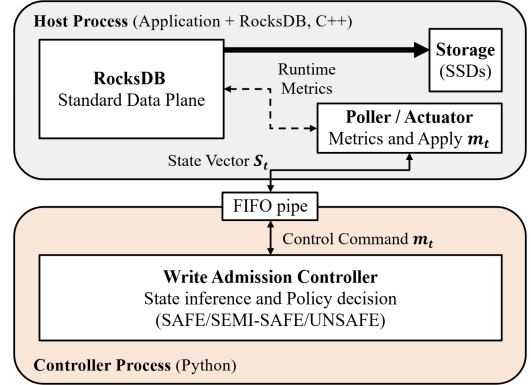


Figure 6: S-RocksDB with in-process poller/actuator and out-of-process controller.

5 CONTROL DESIGN OF S-ROCKSDB

This section presents the control architecture of S-RocksDB, a lightweight control-plane extension to RocksDB that regulates admitted write ingress by adjusting the runtime multiplier m_t . The design strictly follows three principles: (1) *Stall-first safety*: once a stall is observed, immediately prioritize aggressive recovery; (2) *Explicit pre-stall modeling* (i.e., proactive braking): treat the near-stall condition as a distinct, actionable state (SEMI-SAFE) rather than folding it into the SAFE region; and (3) *Safe-region learning encapsulation*: utilize reinforcement learning exclusively in stable regions where exploratory actions cannot trigger systemic collapse.

Operationally, S-RocksDB implements a hierarchical controller: a safety shield for stall-active recovery (UNSAFE), a stability buffer for near-stall regulation with hysteresis (SEMI-SAFE), and a learning-based tuner that optimizes admission only in stable regions (SAFE).

Figure 6 provides a system-level view of S-RocksDB. Foreground writes follow the standard RocksDB data plane (memtable, flush, compaction), while S-RocksDB continuously regulates *admitted* write ingress using the multiplier m_t (Eq. 5–7). To keep overhead off RocksDB’s critical path and support long executions, S-RocksDB separates (i) an in-process poller/actuator (C++) colocated with RocksDB and (ii) an out-of-process controller (Python) that performs state inference and policy decisions.

5.1 Controller Overview

Figure 7 summarizes the S-RocksDB controller workflow. S-RocksDB runs with a periodic control loop, called an epoch. In an epoch, the controller performs:

- (1) **Observe.** It reads a metrics vector that captures internal pressure and stall impact (e.g., L0 pressure, compaction backlog, delayed write rate, stall flags, and tail latency), forming the state observation for control and learning.
- (2) **Classify.** It infers the operational states, $\sigma(t)$, using a stall-first rule and near-stall hysteresis. UNSAFE is triggered by direct stall signals; SEMI-SAFE captures the pre-stall state where pressure approaches critical levels without stall; SAFE denotes stable stall-free operation.
- (3) **Act.** It computes the next admission multiplier using a state-dependent policy: in UNSAFE state, it enforces minimal admission

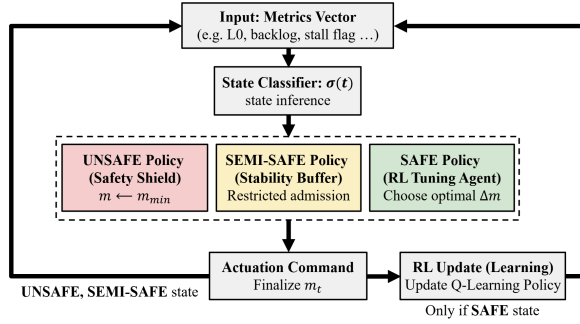


Figure 7: Control loop overview: pressure observation, state classification (σ), and multiplier actuation (m_t).

for recovery, in SEMI-SAFE state, it applies restricted admission to avoid entering stall, and in SAFE state, it uses a Q-learning agent to select an incremental update Δm_t .

(4) **Apply and confirm.** The controller sends the updated m_t to the actuator, which regulates the effective write rate in S-RocksDB.

(5) **Learn (only when in SAFE).** When the system is in SAFE state, the controller updates the Q-learning policy based on the observed reward and the next state.

We use a one-second epoch because key pressure signals (e.g., L0 growth and compaction backlog) evolve meaningfully at this granularity, while shorter intervals can amplify oscillation and increase control-plane overhead.

5.2 Control Knob and Update Constraint

S-RocksDB uses the admission multiplier defined in Eq. (5)–(7). We set $m_{\min} = 0.01$ and treat $m_{\max}$ as a tunable upper bound, and use $m_{\max} = 0.5$ as the default to restrict aggressive admission during long-run training. In our hardware setup, a multiplier of $m_t \approx 0.1 \sim 0.15$ naturally balances the ongoing background compaction rate under the *fillrandom* workload without triggering limits. Therefore, capping it at 0.5 safely trims extreme spikes while covering the entire range of sustainable equilibrium. We test with smaller and larger bounds (i.e., $m_{\max} \in [0.2, 1.0]$ in Section 6.7. We use $m_{\min} > 0$ to avoid full write stoppage by the controller itself; even in recovery, the system continues to make progress while draining backlog.

Further, the upper bound of the per-epoch change is capped by $\Delta m_{\max}$. i.e.,

$$|\Delta m_t| \leq \Delta m_{\max}, \quad (8)$$

where $\Delta m_{\max} = 0.02$ by default.

5.3 State Estimation

5.3.1 UNSAFE: Stall-first detection and post-stall hold. S-RocksDB treats UNSAFE as a stall-active or immediate post-stall recovery condition. The controller enters UNSAFE using direct stall evidence: a write-stop flag, a positive delayed-write rate, or an increase in stall-related counters. Once a stall is observed, S-RocksDB temporarily keeps $m_t = m_{\min}$ for a short post-stall hold window ($T_{\text{hold}} = 10$ s). This hold is not a tuning knob for performance; it is a safety debounce that prevents the controller from immediately returning to aggressive admission while previously accumulated backlog is still draining.

5.3.2 SEMI-SAFE: Near-stall trigger with hysteresis. SEMI-SAFE represents the pre-stall region where foreground writes are not yet blocked, but internal pressure is approaching RocksDB’s native stall margins. S-RocksDB primarily uses two pressure dimensions for this classification: pending compaction backlog $b(t)$ and L0 file pressure $L0(t)$. By default, SEMI-SAFE is entered when compaction backlog becomes high (e.g., $b(t) \geq 32$ GB), backlog rises rapidly, or L0 file pressure indicates an early stall risk (e.g., $L0(t) \geq 16$). These thresholds are placed conservatively below RocksDB’s native slowdown/stop margins (e.g., L0 slowdown around 20 files and backlog entry around 64 GB), so that S-RocksDB reacts before built-in stall mechanisms are triggered.

These thresholds define the safety envelope for state classification, not the fine-grained admission rate. The actual admission adjustment is determined by the SEMI-SAFE step-down policy (Eq. 10) and by the SAFE-region learner after recovery. To avoid state flapping, S-RocksDB also uses hysteresis for SEMI-SAFE entry/release and a hard-lock rule under extreme backlog. These auxiliary constants are debouncing safeguards, not primary policy knobs. Section 6.7 evaluates the dominant threshold parameters and shows that they move the system along predictable throughput–stall trade-offs rather than causing brittle behavior.

5.3.3 Startup guard. At system start, runtime metrics and background throughput can be temporarily unstable due to warm-up and initial compaction. S-RocksDB therefore applies a short startup guard (default: 10 s) that forces $m_t = m_{\min}$. Like the post-stall hold, this guard is a safety debounce rather than a workload-specific tuning parameter.

5.4 State-Dependent Admission Policies

5.4.1 UNSAFE policy: minimal admission for recovery. When in UNSAFE (including the post-stall hold window), the controller prioritizes draining internal pressure:

$$m_{t+1} \leftarrow m_{\min}. \quad (9)$$

5.4.2 SEMI-SAFE policy: restricted admission with cap/floor. In SEMI-SAFE, exploration is disabled and the controller applies a conservative step-down rule:

$$m_{t+1} = \min\left(m_{\text{cap}}, \max(m_{\text{floor}}, m_t - \delta_{\text{semi}})\right). \quad (10)$$

Equation 10 acts as a proactive braking mechanism: it reduces the admission multiplier by a bounded step while keeping it within a safe floor–cap range. The floor prevents the controller from mimicking a full stall, whereas the cap prevents admission from remaining too aggressive near the stall boundary. Thus, SEMI-SAFE serves as a stability buffer between SAFE-region learning and UNSAFE recovery.

5.4.3 SAFE policy: learning-based incremental tuning. In SAFE, the system is stall-free and pressure is low enough to allow cautious performance recovery. S-RocksDB uses an online Q-learning agent to select Δm_t from a small discrete action set:

$$\Delta m_t \in \mathcal{A} = \{-0.02, -0.01, 0, +0.01, +0.02\}, \quad (11)$$

followed by clipping to $[m_{\min}, m_{\max}]$ (Eq. 7) and enforcing $|\Delta m_t| \leq \Delta m_{\max}$ (Eq. 8). This ensures SAFE-region learning cannot make abrupt admission jumps that skip over the near-stall buffer.

5.5 Safe-Region Q-learning

S-RocksDB does not attempt to explicitly “predict” stalls using rigid manual heuristics. The relationship between multi-dimensional ingress traffic, accumulating L0 files, and compounding compaction debt is mathematically non-stationary, highly non-linear, and inherently difficult to model with closed-form convergence guarantees. Under such non-stationary drift, static thresholds or purely theoretical control models inherently fail. Instead of relying on unprovable mathematical assumptions about background capacity, S-RocksDB embraces an empirical and structured approach: the Q-learning agent dynamically discovers and tunes admission in the SAFE region, while physical bounds ($\Delta m_{\max}$) and hysteresis guardrails (SEMI-SAFE) prevent the exploration process from triggering systemic collapse. Crucially, by running strictly out-of-process and evaluating decisions at coarse 1-second epochs, the computational overhead of this continuous learning is negligible and fundamentally separated from RocksDB’s critical I/O paths.

5.5.1 State representation. The Q-learning agent observes a compact continuous state vector s_t (12 dimensions). To ensure the agent captures all facets of S-RocksDB’s health, these features are categorized into three operational domains:

$$s_t = \left\langle \underbrace{L0, b(t), \dots, P99(t), d(t), \dots}_{\text{Internal Pressures}}, \underbrace{}_{\text{QoS / Throughput}}, \underbrace{m_t, t_{\text{safe}}}_{\text{Control Context}} \right\rangle \quad (12)$$

Specifically, the vector includes: current multiplier (m_t), L0 file count, pending compaction and flush backlog ($b(t)$), write ingress, delayed-write rate ($d(t)$), P99 write latency ($P99(t)$), and time since the last UNSAFE episode (t_{safe}). All features are normalized by fixed reference scales and clipped to a bounded range.

5.5.2 Learning and training gate. We use linear function approximation Q-learning:

$$Q(s, a) = \mathbf{w}_a^\top s, \quad (13)$$

$$\delta_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t), \quad \mathbf{w}_{a_t} \leftarrow \mathbf{w}_{a_t} + \alpha \delta_t s_t, \quad (14)$$

with default $\alpha = 0.05$, $\gamma = 0.95$, and ϵ -greedy exploration (default $\epsilon = 0.25$; optionally decayed after warm-up). Since the controller operates continuously without terminal states (a continuing task), we use a discount factor $\gamma < 1$ to ensure convergence.

S-RocksDB updates the Q-learning policy only when the observed reward can be reliably attributed to the action selected by the learning agent. This is not true in SEMI-SAFE or UNSAFE, where deterministic guardrails override the learned action, nor during stall recovery, where the next state is dominated by backlog accumulated before the current action. In addition, because the controller and actuator are decoupled, the measured response may correspond to a previously applied multiplier. These cases can lead to invalid credit assignment if they are used for learning.

To prevent invalid credit assignment, S-RocksDB uses a SAFE-only, actuation-synchronized training gate. The policy is updated only when the system is in SAFE, no stall or post-stall hold is active, SEMI-SAFE is not latched, and the applied multiplier matches the commanded multiplier within a small tolerance. This ensures that each training transition reflects the action actually selected by the learning agent and applied to RocksDB.

5.6 Reward Shaping

The reward prioritizes sustainable long-run admission rather than transient peak throughput. A stall, including delayed-write events detected by the controller, receives a large negative reward. Otherwise, the agent is rewarded for maintaining stall-free execution and higher admitted intensity, while being penalized for abrupt admission changes and rising risk signals. We define:

$$r_t = \begin{cases} -C, & \text{if a stall is observed,} \\ \hat{r}_t, & \text{otherwise,} \end{cases} \quad (15)$$

$$\hat{r}_t = 1 + \lambda_{\text{perf}} \cdot m_{\text{perf}}(t) - \eta |\Delta m_t| - \epsilon_b \min \left(1, \frac{b(t)}{B_{\text{ref}}} \right) - \epsilon_\ell \min \left(1, \frac{\Delta P99(t)}{p_{\text{ref}}} \right). \quad (16)$$

The non-stall reward combines four objectives: surviving an epoch without a stall, encouraging higher admitted intensity in SAFE regions, suppressing oscillatory multiplier changes, and discouraging pressure or latency growth before explicit stalls occur. We set $C = 200$, $\lambda_{\text{perf}} = 1.8$, $\eta = 0.18$, $\epsilon_b = 0.18$, $\epsilon_\ell = 0.15$, $B_{\text{ref}} = 1$ GB, and $p_{\text{ref}} = 1000 \mu\text{s}$. The large stall penalty enforces the intended hierarchy: stall avoidance is primary, pressure stabilization is secondary, and throughput maximization is allowed only within safe operating regions.

5.7 Parameter Roles and Default Selection

S-RocksDB uses several parameters, but they serve different roles. First, safety-envelope parameters, such as SEMI-SAFE backlog/L0 thresholds, $m_{\max}$, and $\Delta m_{\max}$, bound admission aggressiveness and prevent the learner from skipping over the near-stall buffer. Second, learning-adaptation parameters, such as α , γ , and ϵ , determine how quickly the Q-learning agent adapts within SAFE. Third, reward-shaping parameters, such as λ_{perf} and the pressure/latency penalties, express the desired throughput-stability preference once the system is already operating in SAFE.

The defaults are selected as conservative operating points rather than hardware-specific optima. SEMI-SAFE thresholds are placed below RocksDB’s native slowdown/stop margins so that the controller reacts before built-in stall mechanisms are triggered, while the bounded action set and $\Delta m_{\max}$ limit abrupt admission changes.

6 EVALUATION

6.1 Experimental Setup

All experiments were conducted on a single server equipped with two Intel(R) Xeon Gold 6338 CPUs (2.00 GHz), 504 GB of main memory, and a Samsung 870 EVO 1TB/2TB SATA SSD as the primary storage device.

To evaluate, we compare five RocksDB variants: vanilla RocksDB, SILK, ADOC, and two variants of S-RocksDB that differ in background resources and memtable size. **S-RocksDB(R)** uses the same resource configuration as vanilla RocksDB, with 2 background threads and a 64 MB memtable. This resource-matched setting enables a direct comparison of control behavior under the same system capacity as RocksDB. In contrast, **S-RocksDB(S)** uses the same resource configuration as SILK, with 4 background threads and a

128 MB memtable. This SILK-matched setting evaluates whether S-RocksDB remains effective when additional background resources are provisioned.

All systems use their corresponding upstream versions and run with the same workload generator and dataset settings. Table 1 summarizes the versions and key configuration parameters used in our evaluation. In all configurations, BG threads denote the number of background workers used for flush and compaction processing.

Table 1: Configurations used for evaluation.

System	Ver.	Key configuration
RocksDB	10.6.0	2 BG threads; 64 MB memtable
S-RocksDB(R)	10.6.0	2 BG threads; 64 MB memtable; $m_{max} = 0.2$
SILK	5.7.0	4 BG threads; 128 MB memtable
S-RocksDB(S)	10.6.0	4 BG threads; 128 MB memtable; $m_{max} = 0.5$
ADOC	7.7.0	Dynamic control enabled

Workloads and data. We use a write-intensive key-value workload with 16 B keys and 1 KB values. This configuration is chosen to continuously generate flush/compaction pressure and to expose long-run drift effects in the write path.

Metrics. We report (i) write throughput, (ii) stall impact using stalled writes (the cumulative number of foreground writes delayed by RocksDB stall mechanisms), and (iii) tail latency (P99 and higher) to capture user-visible stability under continuous ingestion.

6.2 Microbenchmark

In this section, we conduct a long-running microbenchmark to examine system behavior under continuous write pressure. All systems are subjected to a continuous 24-hour *fillrandom* workload to expose long-run drift and stall emergence under accumulated flush and compaction pressure. We first show the full 24-hour throughput trend, then analyze the stabilized 12–24 hour interval to reveal fine-grained throughput variance. We also evaluate throughput, stall behavior, tail latency (P99–P99.999), and resource usage to assess both performance and stability.

Figure 8 shows throughput over the entire 0–24 hour execution using 10-minute intervals. After the initial 1–2 hour transient phase, the systems form a consistent ordering: RocksDB and S-RocksDB(R) remain in the lower throughput range, SILK and S-RocksDB(S) occupy the middle range, and ADOC stays in the highest range while exhibiting large oscillations. The oscillations are more apparent in Figure 9. The figure shows the fine-grained throughput during the 12–24 hour interval using a 30-second moving average, along with a zoomed-in window. This fine-grained view exposes short-term throughput variance under sustained compaction pressure. Although ADOC achieves the highest average throughput in this interval, 32.38 MB/s, it exhibits the largest fluctuation, with a standard deviation of 62.34 MB/s. Its throughput repeatedly alternates between large bursts and near-zero periods, reaching a peak of 205 MB/s but also dropping to 0 MB/s. RocksDB and SILK show similar instability: they reach peaks of 191 MB/s and 145 MB/s, respectively, but also collapse to 0 MB/s, with average throughputs of 9.48 MB/s and 13.35 MB/s.

In contrast, S-RocksDB provides a smoother throughput trajectory. S-RocksDB(R) achieves 9.66 MB/s with a much lower standard

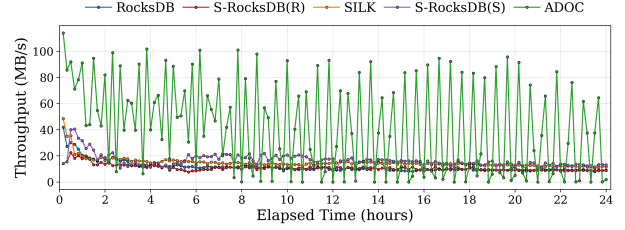


Figure 8: Throughput per 10-minute interval.

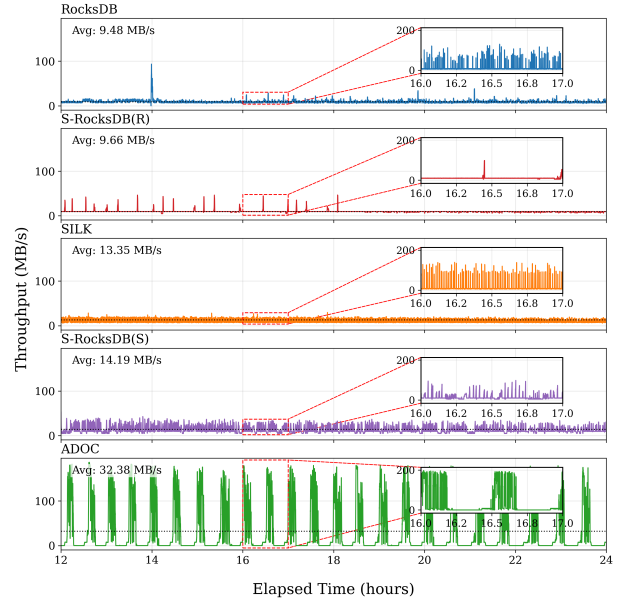


Figure 9: 30-second moving-average throughput during the stabilized 12–24 hour interval, with a zoomed-in view of the 16–17 hour window.

deviation of 4.69 MB/s, while S-RocksDB(S) achieves 14.2 MB/s with a standard deviation of 12.84 MB/s. Although S-RocksDB reaches lower peak throughput than the baselines (98 MB/s for R and 129 MB/s for S), it maintains non-zero minimum throughput throughout the stabilized interval (4.45 MB/s and 4.63 MB/s). These results show that S-RocksDB trades transient throughput peaks for smoother long-run admission and near-zero stalls under sustained write pressure.

Figure 10 reports the stall time in each 1-minute bucket over the full 24-hour execution. This figure provides three observations. First, RocksDB and SILK exhibit high stall time, averaging 49.3s and 45.5s per bucket, respectively. This means that foreground writes are frequently throttled or blocked. Second, ADOC shows a bursty stall pattern. Some 1-minute buckets contain only short stalls, whereas others are fully stalled for the entire 60 seconds. Third, ADOC experiences a prolonged write-stop state lasting 1056.3s. This long stall indicates that foreground Put() requests have been blocked for more than 17 minutes. A reason is that ADOC has no admission policy. Instead, it keeps admitting Put() requests, accumulating compaction pressure. In the longest stall cycle, we

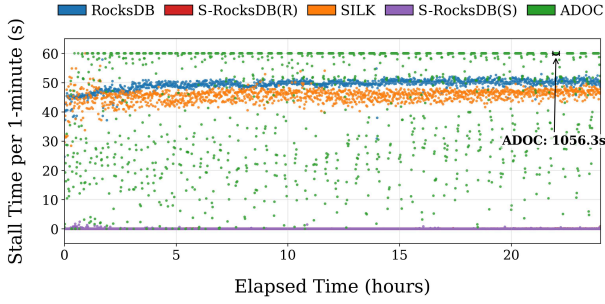


Figure 10: Stall time per 1-minute bucket.

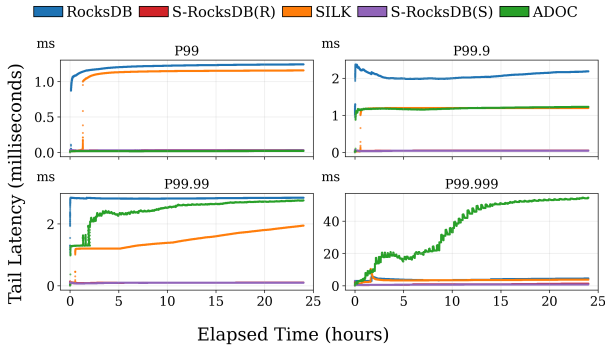


Figure 11: Put() tail latency (P99–P99.999).

observed more than 400 GB pending compaction size. Thus, its high average throughput can still coexist with long intervals in which foreground Put() operations are blocked.

S-RocksDB nearly eliminates such stall behavior. S-RocksDB(R) averages only 0.013s of stall time per bucket, and S-RocksDB(S) further reduces this to 0.007s per bucket. The cumulative stalled-write counts over the full 24-hour run show the same trend. RocksDB, SILK, and ADOC accumulate 64.3M, 555.3M, and 142.9M stalled writes, respectively, whereas S-RocksDB(R) reduces this count to 10,206 and S-RocksDB(S) to only 69. These results demonstrate that S-RocksDB suppresses both persistent and bursty stalls by several orders of magnitude.

Figure 11 reports write tail latency from P99 to P99.999 over the full 24-hour execution. ADOC maintains a low P99 latency, but, as shown in the figure, its P99.999 latency reaches a maximum of 54.54 ms. RocksDB and SILK also remain in the millisecond range at higher percentiles. In contrast, both S-RocksDB configurations keep P99.99 below 0.1 ms and bound the extreme tail to 2.3 ms for S-RocksDB(S) and 3 ms for S-RocksDB(R). As discussed in the previous stall-time analysis, ADOC experiences long write-stop periods; these prolonged stalls appear in the extreme tail and explain its large P99.999 latency.

Figure 12 compares CPU and memory usage over the 24-hour run. Under equal-resource settings, S-RocksDB improves stability without adding background capacity: S-RocksDB(R) uses the same memtable size and background thread count as RocksDB, yet slightly improves stabilized throughput (9.66 MB/s vs. 9.48 MB/s) while reducing stalled writes from 64.3M to 10,206 and lowering average resource usage (98.8% CPU / 2.80 GB to 79.1% CPU / 2.37 GB).

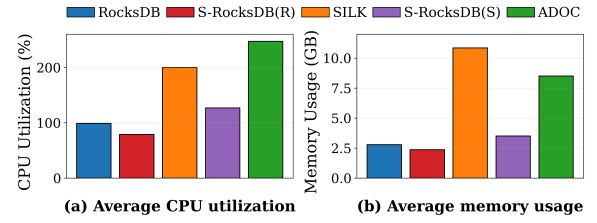


Figure 12: CPU and memory consumption.

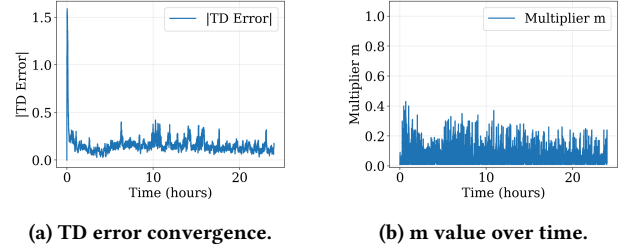


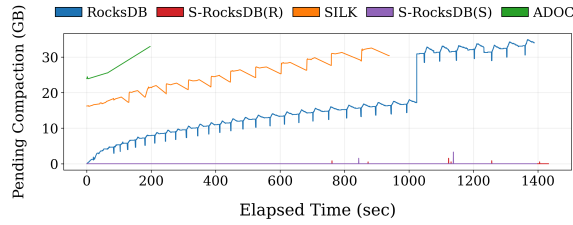
Figure 13: Learning convergence and control trajectory of S-RocksDB(S).

Similarly, S-RocksDB(S) uses the same resource setting as SILK, but improves stabilized throughput (14.2 MB/s vs. 13.35 MB/s), reduces stalled writes from 555.3M to 69, and uses substantially less memory (3.52 GB vs. 10.86 GB). SILK’s high memory usage is mainly due to accumulated pending-compaction backlog, which leads to large compaction jobs and increases memory consumption. ADOC differs from these equal-resource comparisons because it dynamically scales internal resources, up to 512 MB memtables and 20 background threads. This aggressive scaling helps explain ADOC’s high average throughput, but it still does not prevent long stall episodes and incurs higher resource usage (246.98% CPU and 8.52 GB memory) than S-RocksDB(S) (126.85% CPU and 3.52 GB). Thus, S-RocksDB’s stability comes from regulating write ingress before compaction pressure reaches stall-triggering conditions, rather than from simply increasing background capacity.

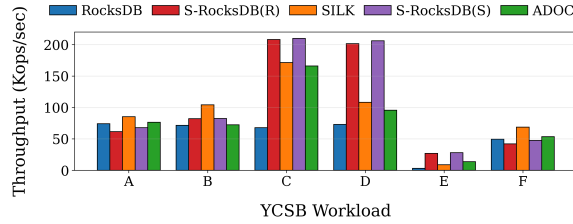
6.3 Learning Behavior and Policy Convergence

A core requirement of S-RocksDB is that online learning remains stable and lightweight. Figure 13a shows that the absolute Temporal-Difference (TD) error [53] is initially elevated due to exploration and nonlinear pressure dynamics, but quickly becomes bounded and remains stable during the rest of the 24-hour run. The Q-gap ($Q_{\max} - Q_{\text{selected}}$) is also small on average (mean 0.056, standard deviation 2.43), indicating that action selection does not suffer from persistent numerical uncertainty.

Figure 13b shows that the admission multiplier settles into a narrow operating band, predominantly between 0.01 and 0.2 and remaining below 0.4, continuously fine-tuning admission to match effective compaction capacity. The controller overhead is negligible: the Q-learning agent consumes only 0.4% CPU on average over the 24-hour execution. These results support that lightweight linear Q-learning is sufficient for this compact admission-control problem and can run off the critical path with minimal overhead.



(a) Pending compaction bytes during the loading phase.



(b) Throughput (Kops/sec).

Figure 14: YCSB benchmark results.

6.4 YCSB Benchmarks

We evaluate S-RocksDB using the standard YCSB [16] workloads A–F, which cover diverse access patterns: A and F include substantial write components, B and C are read-dominant, D emphasizes read-latest behavior, and E introduces short scans. For each workload, we first run the YCSB *load* phase to populate the database and then execute the corresponding *run* phase. This setup allows us to evaluate both ingestion stability during loading and the effect of residual compaction pressure on later workload performance.

Load-phase behavior. Figure 14a shows that RocksDB, SILK, and ADOC retain substantial internal pressure after loading, with 33.99 GB, 33.78 GB, and 32.95 GB of pending compaction, respectively, along with non-trivial L0 file counts. In contrast, both S-RocksDB(S) and S-RocksDB(R) complete the load phase with zero pending compaction bytes and zero L0 files. This indicates that S-RocksDB digests accumulated compaction work before the run phase begins, reducing the residual L0/backlog pressure that can later increase read amplification and metadata traversal.

Read-intensive workloads (B, C, D, E). This lower residual pressure directly benefits read-dominant workloads. In B–E, S-RocksDB delivers comparable or superior throughput to the baselines. In particular, S-RocksDB(R/S) achieves 209K/208K ops/s in workload C and 206K/202K ops/s in workload D, substantially outperforming RocksDB (68K and 73K). Even in scan-heavy workload E, S-RocksDB maintains higher throughput than the baselines. These results show that eliminating load-phase backlog reduces read amplification and stabilizes query performance.

Write-heavy workloads (A and F). For workloads A and F, which contain substantial write or read-modify-write operations, S-RocksDB shows comparable or slightly lower throughput than the most aggressive baselines. This behavior is expected: early in execution, the controller adopts conservative admission to avoid rapid write amplification and unresolved compaction debt. Thus, S-RocksDB may sacrifice short-term write peaks, but it prevents pressure accumulation that later appears as stalls, tail-latency spikes, or degraded read-side performance.

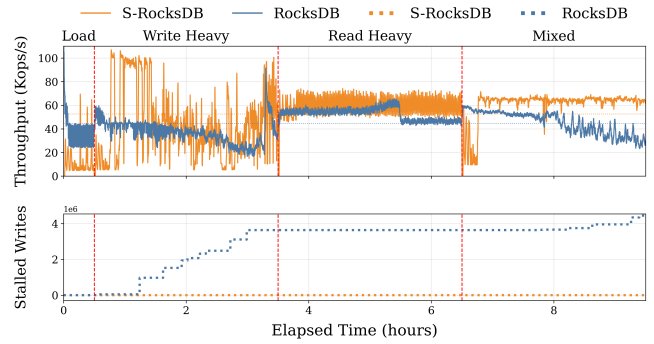


Figure 15: Throughput and cumulative stalled writes under a time-varying workload.

Table 2: Performance comparison across hardware.

HW.	Method	Thr.	Stalls	P99	P99.9	P99.99	P99.999
A	RocksDB	13.8 MB/s	31M	1227 μ s	2020 μ s	2824 μ s	3559 μ s
	S-RocksDB	18.6 MB/s	50	29 μ s	44 μ s	64 μ s	688 μ s
B	RocksDB	18.4 MB/s	30M	1135 μ s	1292 μ s	2724 μ s	2895 μ s
	S-RocksDB	21.6 MB/s	41	24 μ s	34 μ s	75 μ s	469 μ s
C	RocksDB	19.5 MB/s	32M	1137 μ s	2468 μ s	2860 μ s	3906 μ s
	S-RocksDB	19.5 MB/s	17	36 μ s	52 μ s	117 μ s	444 μ s

Overall, the YCSB results show that sustainable admission control not only stabilizes ingestion but also preserves read performance by eliminating residual compaction pressure before the run phase. This confirms that S-RocksDB improves predictability while remaining competitive across diverse workload mixes.

6.5 Robustness across Workloads and Hardware

We next evaluate whether S-RocksDB remains stable beyond a single static workload and a single hardware configuration. Since S-RocksDB(S) shows the strongest overall stability and throughput in the preceding results, we use it as the default S-RocksDB configuration for the remaining experiments unless explicitly stated otherwise. First, we run a time-varying workload with four consecutive phases: load, write-heavy (*r10w90*), read-heavy (*r90w10*), and mixed (*r50w50*). Second, we run 12-hour *fillrandom* experiments on three hardware setups: HW-A uses an Intel Xeon Gold 6338 CPU with a Samsung SATA 870 EVO SSD, HW-B uses the same CPU with a Samsung NVMe 970 EVO Plus SSD, and HW-C uses an Intel i9-9900K CPU with the same NVMe SSD. Across the hardware experiments, all S-RocksDB control parameters are unchanged. Only the bandwidth normalization reference, `write_mb_per_sec`, is set to 500 for SATA SSDs and 3000 for NVMe SSDs; this normalization does not alter the controller logic, safety rules, or learning policy.

Figure 15 shows that S-RocksDB adapts to workload changes while suppressing stall accumulation. RocksDB achieves higher throughput during the initial load phase because it admits writes aggressively, but this creates substantial pressure and leads to 3.6M stalled writes in the write-heavy phase and another 0.8M in the mixed phase. In contrast, S-RocksDB incurs only 103 stalled writes across the entire run. Although S-RocksDB is conservative during load, it achieves higher throughput in all subsequent phases and improves the overall average throughput from 44.6 Kops/s to 52.5

Table 3: Comparison of S-RocksDB with heuristic admission controllers.

Method	Thr.	Stalls	P99	P99.9	P99.99	P99.999
Q-Learning*	18.6 MB/s	50	28.6 μ s	43.6 μ s	64.6 μ s	688.2 μ s
PID	14.5 MB/s	105	31.6 μ s	49 μ s	101 μ s	720 μ s
AIMD	18.3 MB/s	0.63M	30.3 μ s	75.3 μ s	1251.4 μ s	1298.9 μ s
RocksDB	13.8 MB/s	31M	1227 μ s	2020 μ s	2824 μ s	3559 μ s

Note: Q-Learning* denotes the proposed S-RocksDB policy.

Table 4: S-RocksDB(R) with and without SEMI-SAFE.

Setting	Thr.	Stalls	P99	P99.9	P99.99
SEMI On	14.55 MB/s	1	25.2 μ s	42.7 μ s	81.3 μ s
SEMI Off	16.31 MB/s	69,828	28.7 μ s	46.5 μ s	1,194.6 μ s

Kops/s. These results show that regulating admission before pressure reaches the stall region prevents transient write-heavy periods from becoming long-lived stall episodes.

Table 2 further shows that S-RocksDB is not narrowly tuned to one hardware setting. Across all three hardware setups, S-RocksDB reduces stalled writes from tens of millions to only tens of events while matching or improving throughput. Comparing HW-A and HW-B, which use the same CPU but different storage devices, shows that faster NVMe storage increases the sustainable admission rate that S-RocksDB can exploit. Comparing HW-B and HW-C, which use the same NVMe SSD but different CPUs, shows that CPU-side processing capacity can affect background compaction and tail latency. Even so, S-RocksDB consistently lowers P99.99 latency from 2824–2860 μ s to 64–117 μ s and P99.999 latency from 2895–3906 μ s to 444–688 μ s. Overall, the same control structure remains effective across different workload phases, storage devices, and CPUs because the online controller adapts the admission multiplier to observed runtime pressure.

6.6 Controller Baselines and Ablation

We next examine two controller-design questions: whether simpler feedback controllers are sufficient for sustainable write admission, and whether the SEMI-SAFE guardrail is necessary for safe on-line learning. For the first question, we compare S-RocksDB with two heuristic admission controllers: PID (Proportional-Integral-Derivative) [3, 62] and AIMD (Additive-Increase Multiplicative-Decrease) [14, 31]. PID adjusts the admission rate according to the error between observed internal pressure and a target pressure level, whereas AIMD additively increases admission during stable periods and multiplicatively decreases it when pressure approaches a risky region. Both controllers use the same internal pressure signals as S-RocksDB, but they do not use the SAFE-region Q-learning policy or the full SAFE/SEMI-SAFE/UNSAFE control structure.

Table 3 shows the results from 12-hour experiments, indicating that feedback-based admission control is indeed useful, but that the heuristic controllers expose different limitations. PID reduces stalled writes from 31M in RocksDB to 105 and keeps P99.99 latency at 101 μ s, but this stability comes from conservative admission, yielding only 14.5 MB/s throughput. AIMD shows the opposite behavior: it achieves 18.3 MB/s, close to Q-learning’s 18.6 MB/s, but still accumulates 0.63M stalled writes and increases P99.99 latency to 1251.4 μ s. This occurs because AIMD remains reactive: it

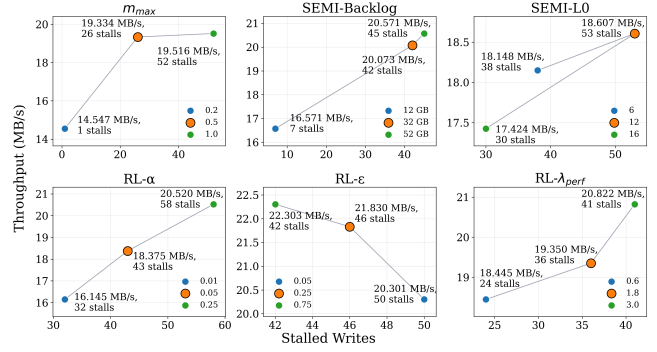


Figure 16: Pareto analysis of control parameters on the average throughput and stalled writes.

increases admission until pressure becomes risky and then reduces the rate only after pressure has already accumulated. In contrast, Q-learning achieves the best balance, reaching the highest throughput (18.6 MB/s) while keeping stalled writes to 50 and maintaining low tail latency.

We then isolate the role of the SEMI-SAFE state, which acts as the pre-stall guardrail between unconstrained SAFE-region learning and stall-active UNSAFE recovery. Table 4 compares S-RocksDB(R) with SEMI-SAFE enabled (SEMI On) against a configuration where the SEMI-SAFE entry logic is disabled (SEMI Off) under a 3-hour *fillrandom* workload. Without SEMI-SAFE, the Q-learning agent continues to explore even as internal pressure approaches critical thresholds. This yields a slightly higher throughput of 16.31 MB/s, but it accumulates 69,828 stalled writes and increases P99.99 latency to 1,194.6 μ s. With SEMI-SAFE enabled, S-RocksDB(R) reduces stalls to a single event and keeps P99.99 latency at 81.3 μ s, at the cost of a moderate throughput reduction to 14.55 MB/s.

Together, these results show that S-RocksDB’s benefit does not come from feedback control alone or from unguarded reinforcement learning. PID is stable but underutilizes the system, AIMD is aggressive but stall-prone, and removing SEMI-SAFE allows the learned policy to overshoot near stall boundaries. The proposed controller avoids these failure modes by combining online learning in SAFE states with deterministic guardrails in SEMI-SAFE and UNSAFE states.

6.7 Sensitivity Analysis of Control Parameters

Overall, S-RocksDB is not brittle to a single parameter setting; the parameters move the system along predictable throughput–stall trade-offs. Figure 16 shows that $m_{\max}$ and the SEMI-SAFE backlog threshold are the dominant deployment knobs. Increasing $m_{\max}$ from 0.2 to 0.5 improves throughput from 14.5 MB/s to 19.3 MB/s with only 26 stalled writes, while increasing it to 1.0 yields little additional throughput (19.5 MB/s) but more stalls. Similarly, raising the backlog threshold from 12 GB to 32 GB improves throughput from 16.6 MB/s to 20.1 MB/s, whereas raising it further to 52 GB gives only marginal gain. The remaining parameters behave interpretably: the L0 threshold has limited impact because SEMI-SAFE uses multiple signals and hysteresis; larger α or λ_{perf} makes the controller more aggressive; and ϵ is less sensitive because exploration is

confined to SAFE. Thus, the defaults are chosen from stable middle regions of the trade-off curves rather than hardware-specific optima, with $m_{\max}$ serving as the primary throughput-stability policy knob.

6.8 Applicability Beyond RocksDB

S-RocksDB is implemented on RocksDB, but its control structure only requires three interfaces from an LSM-tree-based system: observable pressure signals (e.g., SSTable/store-file counts, pending compaction work, flush backlog, or write latency), stall or overload indicators (e.g., blocked writes, backpressure, overload exceptions, or queue saturation), and an admission actuator (e.g., server-side rate limiter, client/proxy-side token bucket, or adjustable write-admission knob). Under this abstraction, RocksDB-specific metrics are one instantiation of the SAFE/SEMI-SAFE/UNSAFE model. Production LSM-tree systems such as Cassandra, HBase, and ScyllaDB expose analogous signals, including SSTable or store-file pressure, pending flush/compaction work, write latency, blocked-update time, compaction backlog, or queue saturation [8, 37, 40, 49, 51]. S-RocksDB could therefore run per node or per shard, with admission enforced at the coordinator, storage node, client driver, rate limiter, or proxy layer. Thus, porting S-RocksDB mainly requires a thin adapter that maps the target system’s pressure metrics and actuation mechanism to the proposed control interface.

7 RELATED WORKS

LSM compaction, write stalls, and runtime control. Prior work mitigates LSM write stalls mainly by improving background execution, reducing interference, or regulating resources. SILK [5] mitigates foreground/background I/O interference, Luo and Carey analyze LSM performance stability under merge policies and I/O budgets [39], and DownForce improves multi-threaded compaction through pipelining [9]. Other systems adapt runtime resources or bandwidth: ADOC harmonizes dataflow using threads and batch sizes [58], RBC controls SSD bandwidth contention [55], RaKV applies token-bucket-style admission for SLA-aware cloud storage [61], and K2/LPNS use scheduling or feedback control at lower system layers [45, 46]. S-RocksDB is complementary to these approaches: rather than redesigning compaction, increasing background resources, or treating the storage engine as a black box, it regulates foreground admission using internal pressure signals and separates SAFE, SEMI-SAFE, and UNSAFE states so that learning is confined to stable regions while risky regions are handled by deterministic guardrails.

Learning-based database and system control. Learning has been widely used in storage and database systems, including learned data structures such as Bourbon, RMI, ALEX, and SageDB [17, 22, 33, 34], SSD tail-latency prediction such as LinnOS [29], and automatic database tuning such as OtterTune [54]. More directly related to LSM-KVSs, RusKey uses reinforcement learning to adjust LSM-tree structures and policies under dynamic workloads [42], while CDBTune and QTune apply Deep Reinforcement Learning (DRL) to database configuration tuning [36, 60]. These systems demonstrate the usefulness of learning for complex system tuning, but they typically target configuration or structural optimization rather than direct, second-scale write admission control.

S-RocksDB deliberately uses lightweight Q-learning instead of a more expressive DRL model [28, 41, 50]. Its admission problem has a compact state representation and a small discrete action space, namely incremental changes to the admission multiplier. In this setting, DRL would introduce neural-network training/inference overhead, additional hyperparameters, higher sample complexity, and less transparent exploration behavior, which are undesirable for an online storage controller that must remain safe and off the critical path. Thus, the novelty of S-RocksDB is not Q-learning alone, but its integration with a storage-specific admission-control abstraction and a safety structure that confines learning to SAFE states while deterministic policies handle SEMI-SAFE and UNSAFE states.

Structural, hardware-aware, and distributed storage. Other work reduces stall risk indirectly by lowering LSM cost or adapting the engine to new hardware. Monkey and Dostoevsky model LSM space-time trade-offs [18, 19]; WiscKey, HashKV, PebblesDB, SlimDB, and LSM-bush reduce write amplification or change LSM organization [11, 20, 38, 47, 48]; and SplinterDB, KVell, SpanDB, MatrixKV, NoveLSM, and ZenFS specialize KV stores for fast or hybrid media [7, 13, 15, 32, 35, 57]. These techniques improve raw service capacity or baseline I/O cost, whereas S-RocksDB controls the time-varying imbalance between foreground ingress and background reorganization without modifying the LSM layout or compaction algorithm. Real-time and distributed KVSs such as In-ConcReTeS and BFT-KVS focus on consistency, deadlines, and fault tolerance [2, 26]; distributed or sharded storage systems [6, 10, 23, 59] can reduce average pressure per shard, but they do not eliminate local LSM imbalance under skewed or hot-shard workloads [1, 43]. S-RocksDB is therefore complementary as a per-node or per-shard admission controller that makes each storage-engine instance more predictable.

8 CONCLUSIONS

This paper reframes write stalls in LSM-KVSs as a continuous admission-control problem rather than a threshold-triggered emergency. S-RocksDB regulates foreground ingestion according to observed internal pressure and uses lightweight Q-learning to track sustainable admission only during stable operation. When the system approaches near-stall or recovery states, deterministic guardrails override learning to prevent exploratory actions from amplifying compaction pressure. Across 24-hour microbenchmarks and workload/hardware robustness studies, S-RocksDB eliminates millions of stall events, compresses extreme tail latency, and sustains predictable throughput without modifying the compaction engine. These results show that write stall instability is not an inevitable cost of high throughput, but a controllable runtime phenomenon.

ACKNOWLEDGMENTS

This work was supported by Institute for Information communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.2021-0-01475, SW StarLab) and the Korea Institute of Ocean & Technology (KIOST) research program (Grant No.PEB0021) and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (No.2022R1A2C1006050).

REFERENCES

- [1] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. 2016. Slicer: {Auto-Sharding} for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 739–753.
- [2] Malte Appel, Arpan Gujarati, and Björn B Brandenburg. 2017. A Byzantine Fault-Tolerant Key-Value Store for Safety-Critical Distributed Real-Time Systems. *CERTS* (2017).
- [3] Karl Johan Åström and Tore Hägglund. 2001. The future of PID control. *Control engineering practice* 9, 11 (2001), 1163–1175.
- [4] Karl Johan Åström and Richard M Murray. 2021. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton university press.
- [5] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhramoorthi, and Diego Didona. 2019. {SILK}: Preventing latency spikes in {Log-Structured} merge {Key-Value} stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 753–766.
- [6] Laurent Bindschadler, Ashvin Goel, and Willy Zwaenepoel. 2020. Hailstorm: Disaggregated compute and storage for distributed lsm-based databases. In *Proceedings of the twenty-fifth international conference on architectural support for programming languages and operating systems*. 301–316.
- [7] Matias Björling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. 2021. {ZNS}: Avoiding the block interface tax for flash-based {SSDs}. In *2021 USENIX annual technical conference (USENIX ATC 21)*. 689–703.
- [8] Dhruva Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, et al. 2011. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 1071–1080.
- [9] Hongsu Byun, Honghyeon Yoo, and Sungyong Park. 2025. Revisiting Multi-threaded Compaction in LSM-trees: Enabling Compaction Pipelining. In *Proceedings of the 54th International Conference on Parallel Processing (ICPP '25)*. Association for Computing Machinery, New York, NY, USA, 794–803. doi:10.1145/3754598.3754675
- [10] Zhichao Cao, Huibing Dong, Yixun Wei, Shiyong Liu, and David HC Du. 2022. Is-hbase: An in-storage computing optimized hbase with i/o offloading and self-adaptive caching in compute-storage disaggregated infrastructure. *ACM Transactions on Storage (ToS)* 18, 2 (2022), 1–42.
- [11] Helen HW Chan, Yongkun Li, Patrick PC Lee, and Yinlong Xu. 2018. {HashKV}: Enabling efficient updates in {KV} storage via hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 1007–1019.
- [12] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Walach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.
- [13] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. {SpanDB}: A fast, {Cost-Effective} {LSM-tree} based {KV} store on hybrid storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 17–32.
- [14] Dah-Ming Chiu and Raj Jain. 1989. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN systems* 17, 1 (1989), 1–14.
- [15] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. {SplinterDB}: closing the bandwidth gap for {NVMe} {Key-Value} stores. In *2020 usenix annual technical conference (usenix atc 20)*. 49–63.
- [16] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.
- [17] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnathan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From {WiscKey} to bourbon: A learned index for {Log-Structured} merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 155–171.
- [18] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 79–94.
- [19] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*. 505–520.
- [20] Niv Dayan and Stratos Idreos. 2019. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data*. 449–466.
- [21] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (2013), 74–80.
- [22] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 969–984.
- [23] Siying Dong, Shiva Shankar P, Satadru Pan, Anand Ananthabhotla, Dhanabal Ekambaram, Abhinav Sharma, Shobhit Dayal, Nishant Vinaybhai Parikh, Yanqin Jin, Albert Kim, et al. 2023. Disaggregating rocksdb: A production experience. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–24.
- [24] Facebook. 2026. RocksDB. <http://rocksdb.org>. Accessed: 2026-02-25.
- [25] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rath, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*. 3–18.
- [26] Arpan Gujarati, Ningfeng Yang, and Björn B Brandenburg. 2022. In-ConcReTeS: Interactive consistency meets distributed real-time systems, again!. In *2022 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 211–224.
- [27] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. 2020. The architectural implications of facebook’s dnn-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 488–501.
- [28] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*. Pmlr, 1861–1870.
- [29] Mingzhe Hao, Levent Toksoz, Nanqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S Gunawi. 2020. {LinnOS}: Predictability on unpredictable flash storage with a light neural network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 173–190.
- [30] Andy Huynh, Harshal A Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2022. Endure: a robust tuning paradigm for LSM trees under workload uncertainty. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1605–1618.
- [31] Van Jacobson. 1988. Congestion avoidance and control. *ACM SIGCOMM computer communication review* 18, 4 (1988), 314–329.
- [32] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning {LSMs} for nonvolatile memory with {NoveLSM}. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 993–1005.
- [33] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A learned database system. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019*.
- [34] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*. 489–504.
- [35] Baptiste Lepercq, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 447–461.
- [36] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2118–2130.
- [37] Junfeng Liu, Haoxuan Xie, and Siqiang Luo. 2026. ArceKV: Towards Workload-driven LSM-compactions for Key-Value Store Under Dynamic Workloads. *Proceedings of the VLDB Endowment* 19, 5 (2026), 958–972.
- [38] Lanyue Lu, Thanumalayan Sankaranarayanan Pillai, Hariharan Gopalakrishnan, Andrea Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. WiscKey: Separating keys from values in ssd-conscious storage. *ACM Transactions On Storage (TOS)* 13, 1 (2017), 1–28.
- [39] Chen Luo and Michael J. Carey. 2019. On performance stability in LSM-based storage systems. *Proc. VLDB Endow.* 13, 4 (Dec. 2019), 449–462. doi:10.14778/3372716.3372719
- [40] Chen Luo and Michael J Carey. 2020. LSM-based storage techniques: a survey: C. Luo, MJ Carey. *The VLDB Journal* 29, 1 (2020), 393–418.
- [41] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *nature* 518, 7540 (2015), 529–533.
- [42] Dingheng Mo, Fanchao Chen, Siqiang Luo, and Caihua Shan. 2023. Learning to optimize lsm-trees: Towards a reinforcement learning based key-value store for dynamic workloads. *Proceedings of the ACM on Management of Data* 1, 3 (2023), 1–25.
- [43] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. 2013. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 385–398.
- [44] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta informatica* 33, 4 (1996), 351–385.
- [45] Sanghyeok Park and Jinkyu Lee. 2021. Analysis of the K2 Scheduler for a Real-Time System with an SSD. *Electronics* 10, 7 (2021), 865.

- [46] Bo Peng, Cheng Guo, Jianguo Yao, and Haibing Guan. 2023. {LPNS}: Scalable and {Latency-Predictable} Local Storage Virtualization for Unpredictable {NVMe} {SSDs} in Clouds. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 785–800.
- [47] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 497–514.
- [48] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *Proceedings of the VLDB Endowment* 10, 13 (2017), 2037–2048.
- [49] Yuanming Ren, Siyuan Sheng, Zhang Cao, Yongkun Li, and Patrick PC Lee. 2026. Holistic and Automated Task Scheduling for Distributed {LSM-tree-based} Storage. In *24th USENIX Conference on File and Storage Technologies (FAST 26)*, 203–219.
- [50] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).
- [51] ScyllaDB. 2026. *Metrics | ScyllaDB Docs*. ScyllaDB. <https://docs.scylladb.com/manual/stable/reference/metrics.html> Accessed: 2026-02-25.
- [52] Richard S Sutton, Andrew G Barto, et al. 2018. Reinforcement learning: An introduction 2nd ed. *MIT press Cambridge* 1, 2 (2018), 25.
- [53] John Tsitsiklis and Benjamin Van Roy. 1996. Analysis of temporal-difference learning with function approximation. *Advances in neural information processing systems* 9 (1996).
- [54] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*, 1009–1024.
- [55] Zepeng Wang and Shu Yin. 2023. RBC: A bandwidth controller to reduce write-stalls and tail latency. In *Proceedings of the 52nd International Conference on Parallel Processing (Salt Lake City, UT, USA) (ICPP '23)*. Association for Computing Machinery, New York, NY, USA, 213–222. doi:10.1145/3605573.3605601
- [56] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8, 3 (1992), 279–292.
- [57] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. {MatrixKV}: Reducing write stalls and write amplification in {LSM-tree} based {KV} stores with matrix container in {NVM}. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 17–31.
- [58] Jinghuan Yu, Sam H. Noh, Young-ri Choi, and Chun Jason Xue. 2023. ADOC: automatically harmonizing dataflow between components in log-structured key-value stores for improved performance. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (Santa Clara, CA, USA) (FAST'23)*. USENIX Association, USA, Article 5, 16 pages.
- [59] Qiaolin Yu, Chang Guo, Jay Zhuang, Viraj Thakkar, Jianguo Wang, and Zhichao Cao. 2024. CaaS-LSM: compaction-as-a-service for LSM-based key-value stores in storage disaggregated infrastructure. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–28.
- [60] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jia Shu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data (Amsterdam, Netherlands) (SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 415–432. doi:10.1145/3299869.3300085
- [61] Yuanhui Zhou, Kai Lu, Zhonghua Wang, Peng Xu, Kai Wang, Ranjun Jia, and Jiguang Wan. 2025. RaKV: A Write-Optimized LSM Store for Cloud Block Storage with Robust SLA. *ACM Trans. Archit. Code Optim.* 22, 4, Article 151 (Dec. 2025), 26 pages. doi:10.1145/3774424
- [62] John G Ziegler and Nathaniel B Nichols. 1942. Optimum settings for automatic controllers. *Transactions of the American society of mechanical engineers* 64, 8 (1942), 759–765.