FlushSync: The Hidden Synchronization Bottleneck in LSM-tree Based Stream Processing

Prajwal Basnet
Augusta University
Augusta, USA
prbasnet@augusta.edu

Shungeng Zhang
Augusta University
Augusta, USA
szhang2@augusta.edu

Jianshu Liu
San Diego State University
San Diego, USA
jliu11@sdsu.edu

Jing Zou
Augusta University
Augusta, USA
jizou@augusta.edu

Abstract

Modern event-driven streaming frameworks (e.g., Apache Flink) rely on auto-scaling heuristics that monitor stage-level metrics such as CPU utilization and backpressure, while remaining agnostic to resource-intensive background activities. We identify *FlushSync*, a distinct performance pathology in Log-Structured Merge-tree (LSM-tree) backends like RocksDB where asynchronous flush operations inadvertently synchronize across subtasks, creating transient resource contention that stalls the critical processing path. Because *FlushSync* is imperceptible to current auto-scalers, it cannot be mitigated by standard horizontal scaling. Through extensive evaluations using the NEXMark benchmark suite, we demonstrate that *FlushSync* is a prevalent bottleneck that severely degrades system throughput. We show that the solution lies in proactive management of "soft resources" within the state backend—specifically, tuning flush and compaction thread allocations to decouple these asynchronous operations and improve throughput. Our work establishes that managing internal state backend contention is a first-order concern for achieving predictable performance in modern streaming applications.

CCS Concepts

• **General and reference** → **Performance**; **Experimentation**; • **Computer systems organization** → *Data flow architectures*; • **Applied computing** → **Event-driven architectures**.

Keywords

streaming processing, soft resource, scalability, performance instability

DEBS '26, Lisbon, Portugal

ACM ISBN 979-8-4007-2693-4/2026/06
https://doi.org/10.1145/3809481.3812612

ACM Reference Format:
Prajwal Basnet, Shungeng Zhang, Jianshu Liu, and Jing Zou. 2026. FlushSync: The Hidden Synchronization Bottleneck in LSM-tree Based Stream Processing. In *The 20th ACM International Conference on Distributed and Event-based Systems (DEBS '26), June 23–26, 2026, Lisbon, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3809481.3812612

1 Introduction

Mission-critical event-driven stream processing applications, such as real-time advertising [27] and anomaly detection [44], operate under stringent latency constraints dictated by live user interactions. Consequently, the industry heavily prioritizes minimizing tail latency (P99), as even infrequent processing spikes can disproportionately degrade the end-user experience [1]. For example, in automated retail environments like Amazon Go [39], customer actions are ingested as continuous event streams to trigger immediate services; a delay of mere seconds can result in immediate customer dissatisfaction and quantifiable revenue loss [17].

To address these latency and throughput demands, the prevailing approach in recent years has heavily favored auto-scaling solutions. These reactive mechanisms primarily target variances in the *critical path* [29], where the direct chain of stage instances that events must traverse to yield a result. However, relying solely on critical-path scaling neglects a severe, hidden source of performance degradation: the asynchronous maintenance path. Building on insights from ShadowSync [47], which demonstrated that background services can drive long-tail latency, we observe that internal state backend dynamics frequently bottleneck scaled systems. In LSM-tree [25] backends, the overlapping execution of asynchronous flush and compaction tasks generates transient "millibottlenecks." This invisible resource contention inherently limits system elasticity, degrading system throughput and inducing Quality of Service (QoS) violations regardless of how many resources are provisioned to the critical path.

Building on recent insights into asynchronous background services, we identify a critical, previously unexplored performance bottleneck caused by the synchronization of RocksDB's flush operations during state checkpointing,

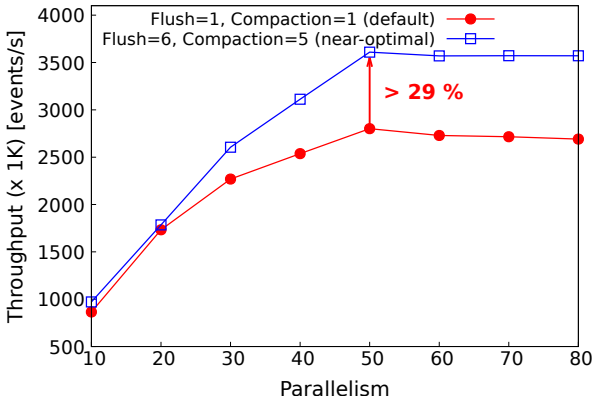


Figure 1: *FlushSync*—synchronization stalls among asynchronous maintenance operations (e.g., RocksDB flushes) caused by inappropriate soft resource allocations—significantly degrades throughput in event-driven stream processing applications.

which we term *FlushSync*. To systematically investigate the impact of *FlushSync* on event-driven stream processing pipelines, we evaluate Flink [35] with a RocksDB [14] state backend on the widely-used NEXMark benchmark [40]. This technology stack is ideal because RocksDB's underlying LSM-tree architecture enforces a strict separation between critical-path and off-path operations. High-velocity data updates are rapidly ingested into in-memory memtables on the critical path, whereas routine maintenance operations—such as flushing full memtables to persistent storage and executing resource-intensive compactions—are relegated to asynchronous background threads. This design provides a near-optimal environment to isolate and study how supposedly non-interfering background tasks actually induce severe resource contention and stall system throughput. Our analysis reveals that the adverse effects of *FlushSync* can be effectively neutralized by proactively tuning "soft resources". Specifically, the thread pools allocated to these internal background operations.

As illustrated in Figure 1, optimizing flush and compaction thread allocations yields a 29% throughput increase for a Flink job (stage degree of parallelism of 50) compared to the default configuration. Conversely, traditional stage scaling (i.e., increasing stage replicas) has a negligible effect on FlushSync-induced stalls. This stark contrast underscores that the careful management of soft resources. Specifically, the thread allocation for background services—is vital for maintaining stable system performance.

Our core contribution is a comprehensive experimental analysis of *FlushSync*. It is generally assumed that offloading tasks from the critical path to asynchronous background services improves system performance [10, 11, 13], a design adopted by modern streaming systems like Flink that offload backend state management to RocksDB. However, our

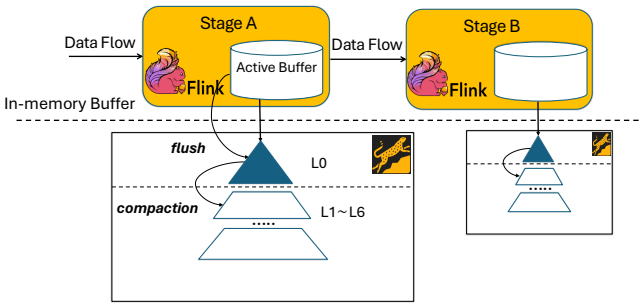


Figure 2: RocksDB implementation of LSM-tree.

extensive experiments demonstrate that shifting work off the critical path introduces hidden coordination overheads. Specifically, the synchronization of flush operations in stateful stage instances substantially degrades overall system throughput by inducing stalls and delaying data propagation. Thus, identifying and mitigating these asynchronous bottlenecks is critical.

Our second contribution is an in-depth analysis of how soft resource allocations impact FlushSync and overall performance. We define soft resources as the configurable system software components that govern hardware utilization. In this context, RocksDB's flush and compaction threads act as the primary soft resources managing state efficiency during checkpointing. By evaluating conservative (low) and liberal (high) allocation strategies, we demonstrate that both intuitive extremes lead to suboptimal performance across varying workloads and storage media.

Finally, our third contribution is the design and implementation of a systematic approach to mitigate FlushSync via near-optimal soft resource tuning. We propose a workload-aware tuning algorithm that intelligently determines the appropriate number of flush and compaction threads, synchronized with Flink stage degree of parallelism. Through extensive evaluations using the NEXMark benchmark suite, our solution effectively suppresses FlushSync, achieving an average throughput increase of approximately 30% across various storage backends.

The rest of the paper is structured as follows. Section 2 characterizes the *FlushSync* bottleneck, while Section 3 analyzes the correlation between soft resource allocations and system throughput. Section 4 details our workload-aware tuning algorithm designed to mitigate *FlushSync*. We then evaluate this solution against default configurations and the ShadowSync baseline in Section 5. Finally, Section 6 discusses the broader applicability of our findings, Section 7 surveys related work, and Section 8 concludes.

2 Background and Motivation

Prior work on improving stream processing performance has largely focused on dynamic scaling techniques [20, 33, 46]

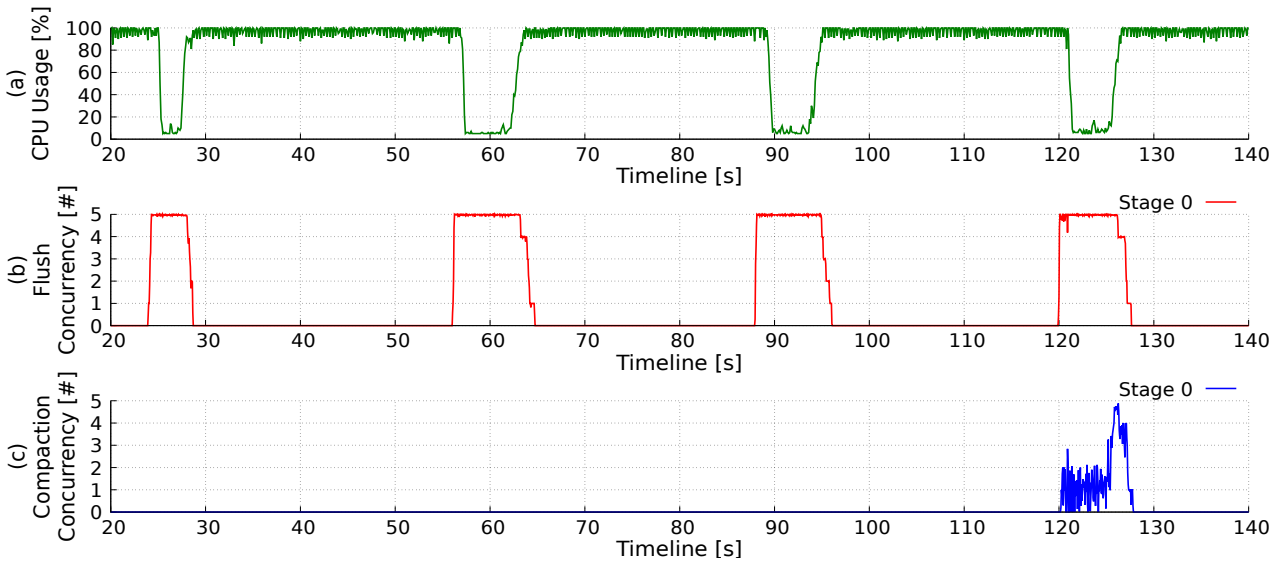


Figure 3: *FlushSync* of flush synchronization for single stage in RocksDB during the checkpointing period, leading to large CPU stalling (e.g., period 24∼28sec, 56∼62sec, 89∼95sec, and 121∼126sec).

that boost throughput by provisioning resources to the critical path. However, a smaller body of work, notably ShadowSync [47], has shown that asynchronous background services, particularly checkpointing for state management [9], can also govern overall system performance. In Flink, checkpointing triggers flush and compaction operations in RocksDB (Figure 2), and the efficiency of these background operations directly affects end-to-end throughput.

Checkpointing ensures data reliability during failures or job restarts. During a checkpoint, Flink captures application state and persists it from the local state backend to remote storage. To do so, it must first persist RocksDB's in-memory memtable to disk. This involves two core RocksDB operations: *flush* writes the memtable contents to Level-0 SSTables on disk; *compaction* later asynchronously merges these SSTables to remove redundant entries and optimize storage layout. Both operations are resource-intensive, demanding substantial CPU and disk I/O. Crucially, flushing is a stop-the-world event [8, 12]—Flink pauses the affected tasks while data is written to disk.

Our experiments revealed a recurring issue: when checkpointing triggers at regular intervals (every 32 seconds), it simultaneously initiates flush and compaction across all stateful stage instances. We instrument state-backend activities at 50 ms granularity to identify and quantify the resulting CPU stalls. RocksDB supports pipelined flushing and parallel compactions [15], and the LSM-tree design enables concurrent background maintenance [28, 32]. However, when stateful stage instances trigger flush operations simultaneously, they contend for a fixed pool of background threads. This contention delays flush completion, forcing Flink tasks to stall. We term this phenomenon *FlushSync*: synchronized flushes across stateful stage instances that create bursts of resource contention, stalling the critical processing path.

Figure 3(a) shows a recurring drop in CPU utilization that coincides with checkpointing events. These drops occur when checkpointing triggers synchronous flush operations across multiple replicas of the same stage, exposing the FlushSync bottleneck. Figure 3(b) illustrates the flush concurrency during execution. In our experimental setup, Flink is deployed in standalone mode on five TaskManager nodes, resulting in five flush threads and five compaction threads, corresponding to the number of TaskManagers. At each checkpointing interval, all flush threads become fully occupied. Since the available flush concurrency is insufficient to serve all pending flush requests immediately, excess flush operations are queued. Because flush operations are synchronous and block normal event-driven stream processing, queued flushes effectively stall the execution, leading to the observed CPU underutilization. Figure 3(c) presents the compaction concurrency. Typically, after about four flushes, a compaction is triggered. Unlike flush threads, compaction threads are not fully saturated. While compaction can be resource-intensive, its performance impact varies with workload characteristics, disk behavior, and system configuration. Even with sufficient hardware resources, background tasks such as flush operations can still affect overall performance. In this setup, the observed degradation was primarily linked to FlushSync-induced synchronization delays.

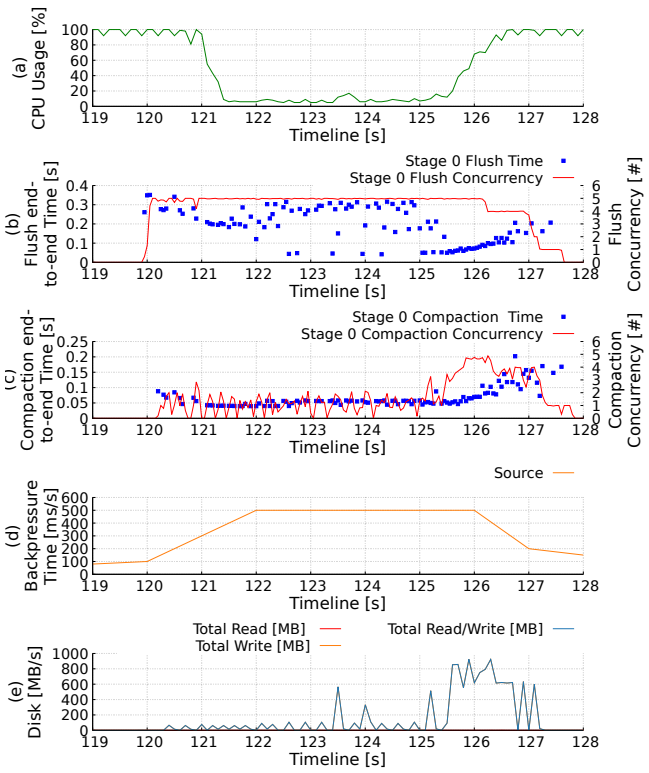


Figure 4: Zoom in analysis of *FlushSync* event between 119~128sec, for single stage in RocksDB showing the relationship between CPU usage, flush and compaction activity, backpressure, and disk I/O. The figure illustrates how synchronized flush and compaction operations lead to cascading system effects.

To understand this synchronization, we zoom in on the 4th checkpoint (119~128sec) in Figure 4. Figure 4(b) shows that most flush operations take 0.2–0.3sec to complete. Flush concurrency rises from 119.9sec and stabilizes at 121sec with all 5 flush threads fully occupied, causing CPU utilization to drop sharply, (Figure 4(a)) as the pipeline stalls waiting for flushes to finish. Compaction (Figure 4(c)) initially completes quickly (~0.05sec), but between 125.8~126.2sec all 5 compaction threads become saturated and end-to-end compaction times increase. Unlike flush, compaction is CPU-intensive, so CPU utilization begins to recover during this phase. Flink reports backPressuredTimeMsPerSecond as the number of milliseconds per second that a task is unable to emit records [3]. As flush contention grows, backpressure at the source rises to ~500 ms/s (Figure 4(d)), meaning the source is blocked for roughly half of each second. Disk read/write throughput spikes between 125.5–126.6sec (Figure 4(e)), coinciding with peak compaction activity. This zoom-in reveals that FlushSync is not a simple synchronization of flush operations. It triggers longer end-to-end processing times for both flush and compaction, creating a cascading effect across CPU, disk, and pipeline flow that significantly degrades performance.

Addressing FlushSync is therefore critical to optimizing performance. Simply scaling hardware resources (e.g., adding task managers) [33] does not resolve the issue, as the bottleneck stems from internal synchronization stalls rather than resource scarcity. Instead, effective tuning of soft resources—specifically, the thread pools allocated to background flush and compaction operations—is essential to fully utilize existing infrastructure.

3 Experimental Study of FlushSync

We use the NEXMark [40] online auction benchmark to study how soft resources affect Apache Flink with RocksDB, and show that tuning stage parallelism and flush/compaction thread counts can mitigate FlushSync-induced stalls.

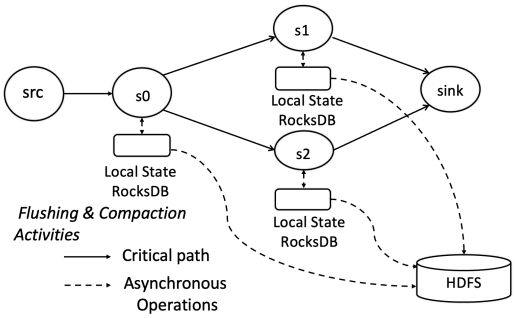
3.1 Experimental Environment

All experiments were carried out on the CloudLab [37] public cloud platform. Our setup included one master node and five task manager nodes.

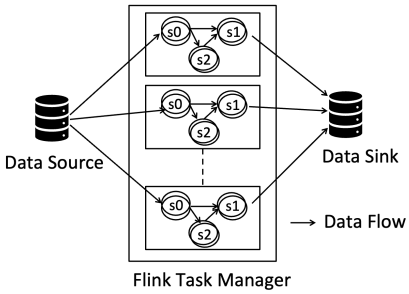
We use NEXMark [40], an open-source real-time online auction system, as our benchmarking tool. To demonstrate our evaluation, we selected Query `q8` from the NEXMark suite. This query tracks new users who create auctions within the last 10 seconds, with the results refreshed every 10 seconds. For this experiment, we configured the workload distribution as follows: 52% bids, 36% auctions, and 12% person events. This setup emphasizes the creation of auctions and person-related records, both of which are handled as stateful data. The system ingests data at a high throughput of 4 million events per second, with a total of 800 million events processed during the experiment. As shown in Figure 5a, Query `q8` is composed of three stateful stage instances, which we label as Stage 0, Stage 1, and Stage 2. Each stage maintains its local state using RocksDB, Flink's default embedded key-value store. RocksDB handles state persistence and performs asynchronous background activities such as flush and compaction, which are internally managed and can affect performance. All synthetic input data is generated using Flink's built-in datagen source [4], which acts as the pipeline's source stage. The overall benchmarking topology is illustrated in Figure 5b, where data is generated by the datagen source, processed through multiple stages, and ultimately written to a sink. The hardware and software configuration details are provided in Figure 5c.

3.2 Impact of Stage Parallelism

In our experiments, we studied how changing the degree of parallelism of stateful stage instances affects the performance of Flink. Our setup includes three stateful stage instances



(a) Pipeline flow of Query q8.



(b) Benchmark cluster topology.

Hardware Stack	
Node	c220g5
CPU	Two Intel Xeon Silver 4114 10-core CPUs at 2.20 GHz
RAM	192 GB ECC DDR4-2666 Memory
Disk	One 1 TB 7200 RPM 6G SAS HDD One Intel DC S3500 480 GB 6G SATA SSD
NIC	Onboard Intel i350 1Gb
Software Stack	
Flink Server	Apache Flink 1.17.2
Operating System	CentOS Stream 8 (Kernel 4.18.0)
State Backend Database	RocksDB 6.20.3
Flink Application	NEXMark
JDK Version	Oracle JDK 1.8

(c) Hardware and software stack.

Figure 5: Benchmark setup for Query q8, including the pipeline flow, cluster topology, and hardware/software stack.

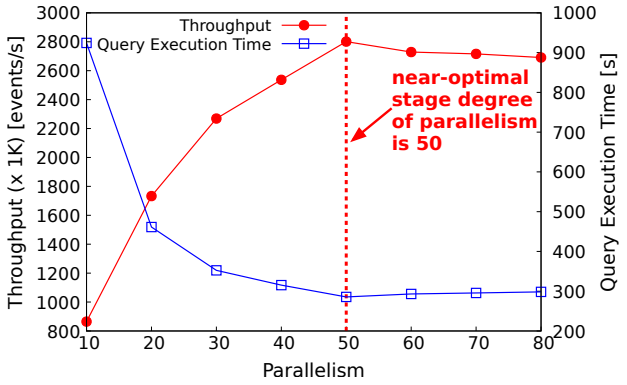


Figure 6: Impact of Flink degree of stage parallelism on the throughput and query execution time of NEXMark query q8. The best degree of parallelism is 50 in our environment.

Stage 0, Stage 1, and Stage 2, running with RocksDB on SSD storage. To improve the performance of Flink, the first step is to scale the critical path stage instances in the pipeline. Each stage should have a degree of parallelism that matches the workload it's handling. We gradually increased the **stage degree of parallelism** (i.e., the number of instances each stage runs) to see how it impacts throughput and query execution time. From the results shown in Figure 6, we found that increasing the stage degree of parallelism generally helps Flink perform better, but only up to a point. Specifically, performance improved steadily until we reached 50 degree of stage parallelism, where we observed the highest throughput and the fastest execution. However, an increasing stage degree of parallelism beyond this point actually caused performance to drop.

After 50 degree of stage parallelism, we can analyze that there is a negative impact on the Flink performance, which is due to the accumulated backpressure on the Flink stage pipeline chain. Some downstream stage instances are having congestion and which slows down the overall performance of Flink. This decline in Flink performance happens for a few reasons. First, too much stage degree of parallelism can cause backpressure in the pipeline, where some parts of the system get overloaded and slow everything else down. Second, as more stage instances perform flush operations in RocksDB at the same time, disk I/O gets overwhelmed, causing delays in checkpointing and data writing.

While the stage degree of parallelism is an essential tuning knob for Flink performance, blindly increasing it does not guarantee improved throughput or faster execution. Instead, the relationship is non-linear, and the system reaches a near-optimal stage degree of parallelism, where the benefits of parallel execution are balanced with the cost of coordination, I/O, and resource contention. So while it might seem like an increasing stage degree of parallelism always boosts performance, our findings show this isn't the case. There is a near-optimal spot, 50 in our experiment, where the workload is well-balanced, and the system runs at its best. Going beyond that adds more overhead than benefit. This shows how important it is to find the right degree of stage parallelism level that matches the workload and available hardware instead of assuming more is always better.

3.3 Impact of RocksDB Flush and Compaction Threads

In this section, we show the impact of flush and compaction activities on Flink performance. We note 1) flush in RocksDB will lock the in-memory state (memtable) of each stage instance and prevent further state update operations, blocking the normal message processing in the critical path of the streaming dataflow, and 2) compaction is inherently an asynchronous process but could contend for the shared CPU resources with message processing threads. We start with the default configuration of flush and compaction threads in RocksDB (per Flink TaskManager node), which is one flush thread and one compaction thread. **We set the stage degree of parallelism to 50** and then monitor the behavior of flush and compaction activities during continuous checkpointing.

In Figure 7(a), the end-to-end latency for each completed checkpoint is shown. All checkpoint latencies are within the

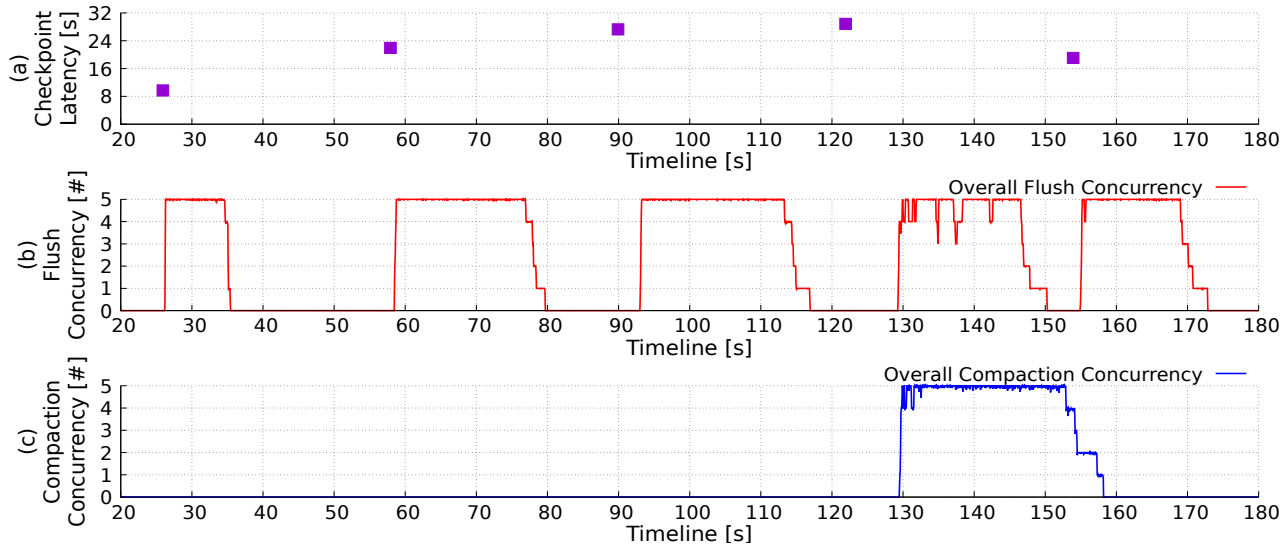


Figure 7: The synchronization of bursts of background asynchronous activities caused by checkpointing from multiple stateful stage instances for default soft resources of flush and compaction threads.

configured 32-second interval. Specifically, the first checkpoint begins at 25 seconds, followed by subsequent checkpoints at 57, 89, 121, and 153 seconds, with 32 seconds apart as expected. Figure 7(b) shows the overall flush concurrency across all stateful stage instances in the Flink pipeline, providing insight into how flush activity is distributed during these intervals. Each checkpoint triggers a flush operation across all stateful stage instances simultaneously. During the period between 20 and 180 seconds, we observe five distinct flush cycles occurring at regular intervals: the first from 25 to 35 seconds, the second from 59 to 80 seconds, the third from 93 to 118 seconds, the fourth from 130 to 150 seconds, and the fifth from 155 to 172 seconds. During these flush windows, all available flush threads become active. However, due to the limited number of threads, some flush operations are forced to wait in a queue, resulting in spikes in latency. Figure 7(c) shows the compaction concurrency associated with these flush events. By default, four consecutive flushes trigger one compaction. In this case, compaction activity is observed between 130 and 159 seconds. Compaction in RocksDB is initiated when four SST files accumulate at level 0, at which point the database merges and reorganizes the data to eliminate redundancy. This process essentially performs de-duplication, compressing the data into more efficient chunks and helping maintain the performance and manageability of the state backend. In RocksDB, the memtable or write buffer is used to temporarily store in-memory data before it's flushed to a level-0 SST file on disk.

FlushSync Problem. By default, RocksDB provisions only two concurrent background jobs: one flush thread and one compaction thread. During a checkpoint, all stateful stage instances must flush their in-memory memtables to Level-0 SST files on disk. Because flushing is a stop-the-world event [8, 12], Flink tasks block until the flush completes. With three stateful stages each running 50 parallel subtasks, up to 150 RocksDB instances may require a flush simultaneously at a single checkpoint. However, each TaskManager provides only one flush thread, forcing the majority of flush requests to queue. This queuing creates backpressure that propagates upstream—stalling the pipeline and causing CPU utilization to collapse as tasks sit idle waiting on the single flush thread to drain the backlog. The default background thread configuration thus becomes a severe bottleneck, independent of hardware capacity.

Impact of Flush and Compaction Threads in RocksDB on Flink Performance.

As we've seen, FlushSync can significantly hinder Flink's performance. In this subsection, we explore how allocating near-optimal soft resources—specifically, flush and compaction threads in the local state backend—can help mitigate this issue. Our focus is on evaluating the impact of increasing the number of compaction threads while keeping the flush thread count at its default value. By doing this, we aim to understand how tuning compaction thread allocation affects Flink's performance. Figure 8 demonstrates that increasing the compaction thread count to 5 results in an 11% improvement in throughput compared to the baseline. Similarly, Figure 8 shows that this configuration also reduces the query execution time by 27 seconds. These results indicate that tuning compaction thread allocation can lead to noticeable performance gains in Flink.

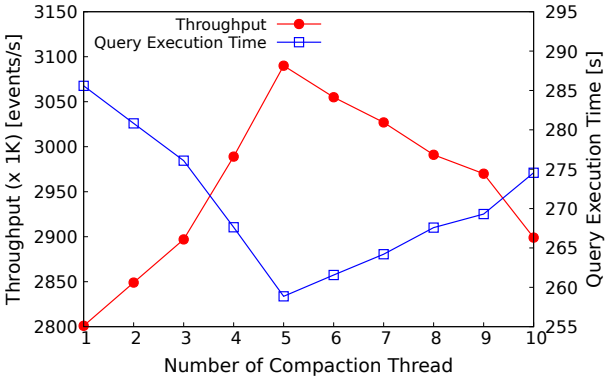


Figure 8: Impact of number of compaction threads on the throughput and query execution time of NEXMark q8. The best compaction thread allocation is 5 in our environment.

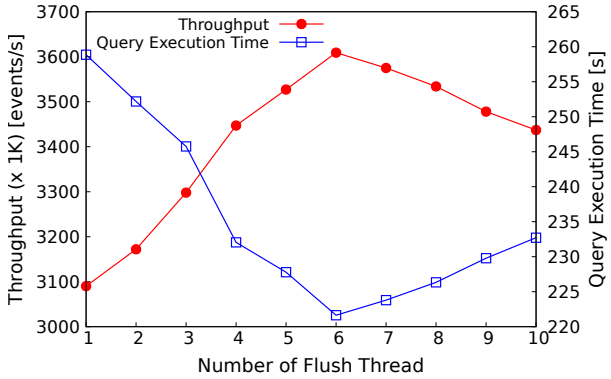


Figure 9: Impact of number of flush threads on the throughput and query execution time of NEXMark q8. The best flush thread allocation is 6 in our environment.

After identifying that using 5 compaction threads yields the best performance for Flink, we used this configuration as a baseline to explore the impact of the number of flush threads. Optimizing not just the compaction threads but also the flush threads is essential for achieving both high performance and efficient resource utilization. When flush operations complete quickly, the risk of performance bottlenecks caused by FlushSync is greatly reduced. Therefore, selecting the right number of flush threads is critical and should be tailored to the specific workload.

Our experiments show that tuning flush threads has a significant effect. As shown in Figure 9, increasing the flush thread count to 6 results in a 17% improvement in throughput. Additionally, the query execution time is reduced by 37 seconds. By allocating near-optimal thread resources for both compaction and flush, we observed a significant improvement in Flink's performance. In this set of experiments, we determined that configuring 6 *flush threads* and 5 *compaction threads* provides the best results for our workload. As shown in Figure 1, this configuration led to a 29% increase in throughput compared to the default settings. These results emphasize the importance of fine-tuning soft resource parameters to optimize performance in LSM-based stateful streaming applications like Flink.

4 Mitigation Solution

So far, we have studied the impact of soft resources on the performance of a stateful Apache Flink application using RocksDB as its state backend. In this section, we present a systematic offline approach to mitigate the identified performance issues by tuning key soft resource parameters. Our goal is to optimize the configuration of these parameters to enhance overall system performance.

We adopt an offline tuning approach to avoid the risks inherent in online exploration. In latency-sensitive event-driven streaming systems, exploratory actions such as testing suboptimal flush or compaction configurations can trigger cascading backpressure and SLO violations. Furthermore, the high variance of runtime metrics in stateful streaming workloads makes online convergence slow and unreliable. Our offline methodology enables safe, reproducible policy learning under controlled conditions, producing validated configurations that can be deployed deterministically without runtime overhead.

Algorithm 1 Near-Optimal Soft Resource Configuration

Require: Flink Job Pipeline J, Stages $S = \{s_0, s_1, \ldots, s_n\}$, Configuration Ranges C, Evaluation Metric M ▷ *Throughput, Execution Time*
Ensure: Near-Optimal Soft Resource Configuration O
Initialize empty list R ▷ *Performance per config*
for $p \in C_{parallelism}$ **do**
  Apply p to all stages in J
  Run workload and measure M
  Record result (p, M) into R
end for
$p^* \leftarrow \arg\max_p M$
for $c \in C_{compaction}$ **do**
  Apply p^*, and c to J
  Run workload and measure M
  Record result (c, M) into R
end for
$c^* \leftarrow \arg\max_c M$
for $f \in C_{flush}$ **do**
  Apply p^*, c^* and f to J
  Run workload and measure M
  Record result (f, M) into R
end for
$f^* \leftarrow \arg\max_f M$
$O \leftarrow (p^*, c^*, f^*)$
return O

We target three critical soft resource parameters: the degree of Flink Stage Parallelism, and the number of RocksDB flush and compaction thread pools. These parameters collectively govern the efficiency of state persistence and the

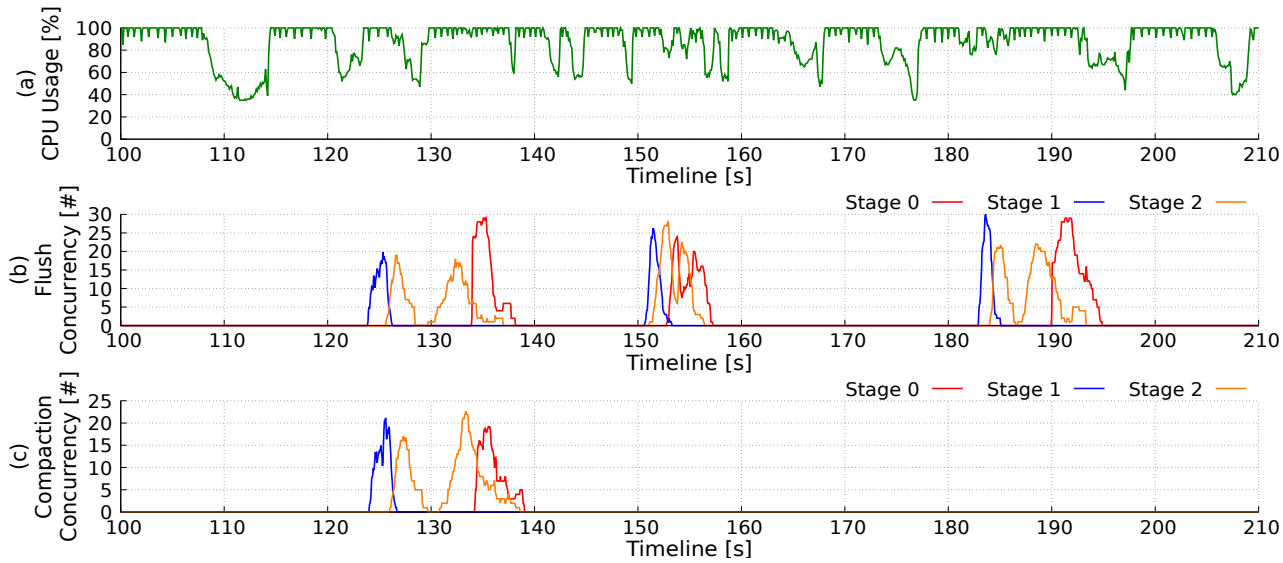


Figure 10: Mitigation of *FlushSync* in RocksDB's flush and compaction activities across stateful stages for Query q8, demonstrating how near optimized soft-resource allocation reduces synchronization stalls and improves overall resource utilization.

severity of resource contention during checkpointing. We follow a structured, step-by-step tuning approach targeting these key soft resource parameters, as shown in Algorithm 1:

- **Flink Stage Parallelism** (lines 2 to 6): This parameter dictates the logical concurrency of the dataflow. While an increasing stage degree of parallelism theoretically scales compute capacity, it simultaneously increases the number of distinct RocksDB instances, thereby intensifying contention for shared physical resources (CPU and I/O bandwidth).
- **RocksDB Compaction Threads** (lines 7 to 12): This setting controls the concurrency of background compaction—the process of merging SSTables to reclaim space and optimize read/write amplification. The number of threads determines the system's ability to sustain high write throughput without triggering write stalls from the accumulation of uncompacted files.
- **RocksDB Flush Threads** (lines 13 to 18): This parameter governs the concurrency of persisting immutable memtables to storage. Adequate provisioning of flush threads is essential to prevent *FlushSync*, particularly during high-load intervals synchronized by checkpoint barriers.

5 Experiment Evaluation

In this section, we provide a more detailed analysis of *FlushSync* and its impact on Flink's performance when using different types of disks (Section 5.1)—specifically, in SSDs, traditional HDDs, and NVMe drives—as the local state backend and when running diverse query workloads (Section 5.2).

5.1 Impact of Storage Hardware on State Backend Performance

5.1.1 Evaluation with SSDs as State Backend. We conducted our evaluation using query q8 from the NEXMark benchmark suite, configured to process 4 million events per second for a total of 800 million events. The checkpointing interval was set to 32 seconds. All experiments were carried out on the CloudLab public cloud platform [37], using a cluster with one master node and five task manager nodes. The hardware specifications and the corresponding software configurations are listed in Figure 5c. For this experiment, RocksDB was configured to use SSDs as the local state backend.

After applying our mitigation algorithm—which allocates adequate soft resources for flush and compaction—we observed a significant improvement in Flink's performance (as shown in Figure 1). In Figure 10(a), the CPU usage stays high—around 90% on average—which means the system is using its hardware much more efficiently, with fewer idle moments or slowdowns. Figure 10(b) further displays the flush activity for *Stage 0, Stage 1, and Stage 2*. Before tuning, flush operations from these stages often happened at the same time, which caused a slowdown due to resource contention. "FlushSync" made Flink tasks wait for disk operations to finish. Figure 10(c) shows that compaction activities are not highly time-consuming, and contention for compaction threads is also alleviated through sufficient and near-optimal resource allocation. After tuning the number of flush and compaction threads, the flush jobs still happen regularly, but they are better spaced out.

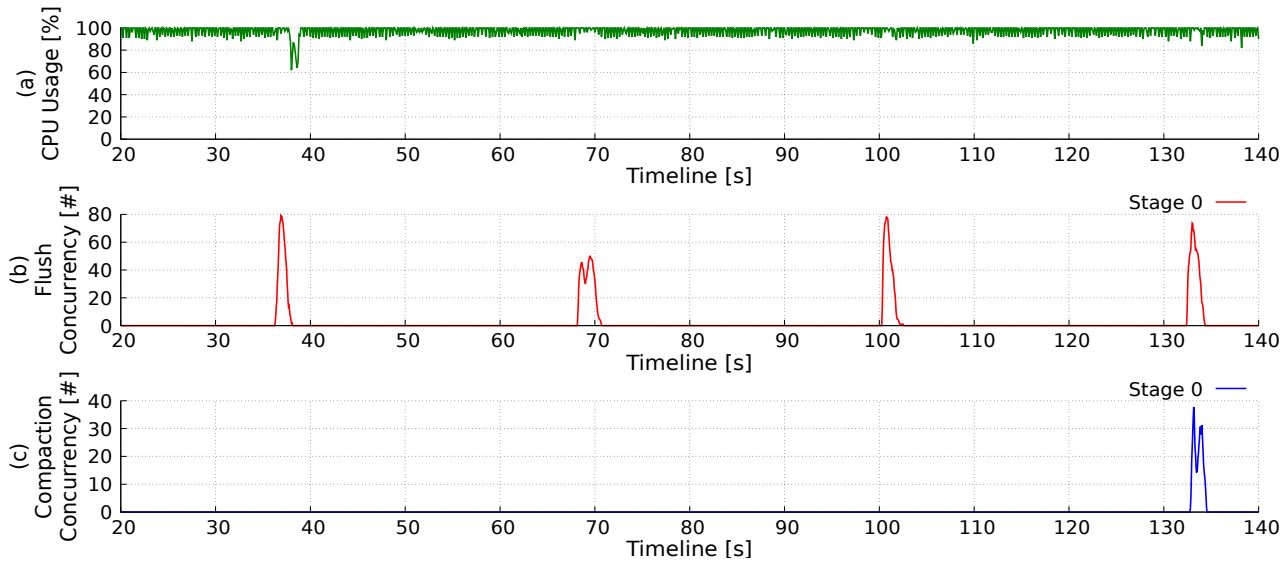


Figure 11: Mitigation of *FlushSync* induced by RocksDB's background flush and compaction operations for Query q12 with single stateful stage.

To simplify the Flink job pipeline, we select NEXMark query q12, which contains a single stateful stage. We evaluate our mitigation approach using this single-stage workload. Figure 11(a) shows that CPU utilization remains stable, with no idle valleys, after allocating sufficient flush threads. Unlike the baseline configuration, FlushSync no longer induces CPU stalls. Figure 11(b) illustrates flush activity synchronized with periodic checkpointing at 32-second intervals. With workload-aware flush-thread allocation, flush operations complete without being queued. As a result, stream processing continues smoothly without waiting for flush operations to finish. Figure 11(c) shows that compaction activity remains unsaturated and does not become time-consuming. Although compaction is a resource-intensive task, it runs asynchronously in the background and does not interfere with foreground data processing, as reflected by the stable CPU utilization and non-blocking flush behavior. Overall, these improvements demonstrate that properly setting internal thread counts for flush and compaction helps Flink run faster and more efficiently—without needing extra hardware.

5.1.2 Evaluation with HDDs as State Backend. Using the same workload and checkpoint interval as in the SSD experiments (Section 5.1.1) for query q8, we simply replaced the storage medium with HDDs. We compared Flink's processing performance using the **default configuration** of flush and compaction threads versus using the **near-optimized thread allocation**. As shown in Figure 12, optimizing these soft resources led to substantial improvements in throughput across different degree of stage parallelism levels. Specifically, we observed a 28% increase (an improvement of 699K

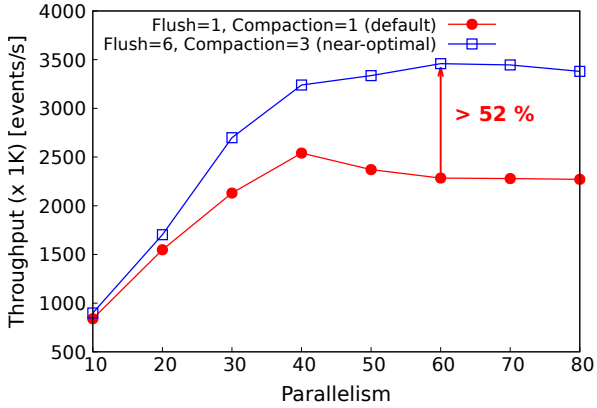


Figure 12: Performance with different degree of parallelism of stateful stage instances on HDDs as state backend in Flink.

events/sec) at 40 degree of stage parallelism, a 41% increase (965K events/sec) at 50 degree of stage parallelism, and a 52% increase (1175K events/sec) at 60 degree of stage parallelism. These results emphasize a significant enhancement in Flink's performance when near-optimal thread settings are applied.

Under the default soft resource configuration, we observed that Flink achieves its highest throughput at 40 degree of parallelism. However, beyond this point, performance begins to decline. This drop is caused by accumulated backpressure within the stage pipeline—particularly in downstream stage instances, where congestion builds up and slows the overall data flow, ultimately degrading Flink's performance. Analysis of query performance reveals that optimizing flush and compaction thread configurations leads to higher throughput compared to the default setup. Overall finding shows that

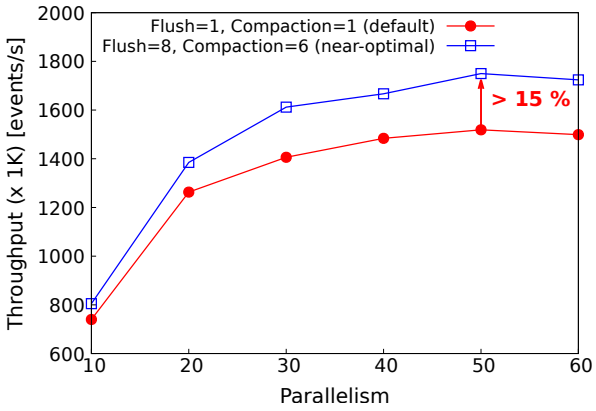


Figure 13: Performance with different degree of parallelism of stateful stage instances on NVMe as state backend in Flink.

balanced soft resource allocation can significantly enhance system efficiency under a high degree of parallelism. This demonstrates that soft resource tuning—specifically, configuring the right number of flush and compaction threads—can significantly improve performance and efficiency of LSM-based stateful streaming engines like Apache Flink.

5.1.3 Evaluation with NVMe as State Backend. While the baseline experiments were conducted on `c220g5` nodes, as shown in Figure 5c, the NVMe-specific evaluation requires local NVMe storage, which is not available in that configuration. Therefore, for this set of experiments, we use CloudLab `m510` nodes (equipped with 256 GB NVMe flash storage). We conducted our experiments using query `q8` from the NEXMark benchmark suite, with an ingestion rate of 2 million events per second and a total of 400 million events, configured with a checkpoint interval of 32 seconds. The cluster setup consisted of one master node and five task manager nodes.

We compared the performance of Flink processing using the default configuration of flush and compaction threads against a setup with near optimally tuned thread allocations. As shown in Figure 13, optimizing these soft resources resulted in a 15% improvement in throughput, an increase of 231K events/sec at 50 degree of stage parallelism. These gains represent a meaningful boost in Flink's processing efficiency. Based on the default configuration, 50 degree of stage parallelism emerged as the most effective setting for the Flink pipeline. When evaluating query performance, the benefits of near-optimal thread tuning become clear. The optimized configuration yields higher throughput at 50 degree of stage parallelism, demonstrating more efficient resource utilization. These results demonstrate that careful tuning of soft resources—particularly the configuration of flush and compaction threads can substantially enhance throughput and overall efficiency in LSM-based stateful streaming systems.

Table 1: Nexmark queries [40]. The number of stateful stage instances in each query, with their descriptions along with input ingestion transaction per second for different disk type in RocksDB State Backend.

Query	Description	Stages	Input rate (events/s)	
			SSD/HDDs ($\times 10^3$)	NVMe ($\times 10^3$)
q7	**Highest Bid**: monitors the highest price items currently on auction	2	1000	500
q8	**Monitor New Users**: identify active users in last period	3	4000	2000
q11	**User Sessions**: compute number of bids each user makes while active	1	4000	2000
q12	**Processing Time Windows**: compute number of bids a user makes within a fixed processing time	1	12000	6000

5.2 Performance Across Diverse Queries

We evaluate our strategy using three representative NEXMark queries (`q7`, `q11`, and `q12`), summarized in Table 1, chosen for their diverse windowing semantics and state behaviors. Specifically, `q7` finds the highest bid per tumbling window using windowed and join states. `q11` counts user bids across sessions, creating demanding RocksDB state interactions at high event rates. `q12` counts bids within 10-second processing-time windows. Despite simple Flink execution graphs, RocksDB's asynchronous background operations (flushes and compactions) still caused noticeable performance degradation.

The queries were tested under high-throughput workloads to simulate realistic streaming conditions. Input data was synthetically generated using Flink's built-in datagen source [4], with a skewed distribution of 92% bid, 6% auction, and 2% person events, ensuring a bid-dominant stream that stresses state access and update operations. For Query `q7`, the ingestion rate was set to 1 million events per second for SSDs and HDDs disk type as the state backend, and 500 thousand events per second for NVMe. For query `q11`, the system ingested 4 million events per second using SSDs and HDDs, and 2 million per second when using NVMe. Query `q12` featured a more aggressive ingestion rate of 12 million events per second on SSDs and HDDs, and 6 million per second on NVMe. To ensure consistency across all experimental setups, the checkpointing interval was fixed at 32 seconds for all storage backends. All experiments were conducted on a five-node Flink cluster, with each node configured as a dedicated task manager.

We compare the default configuration against a ShadowSync inspired mitigation approach and our optimized approach. ShadowSync introduces a delay to prevent flush

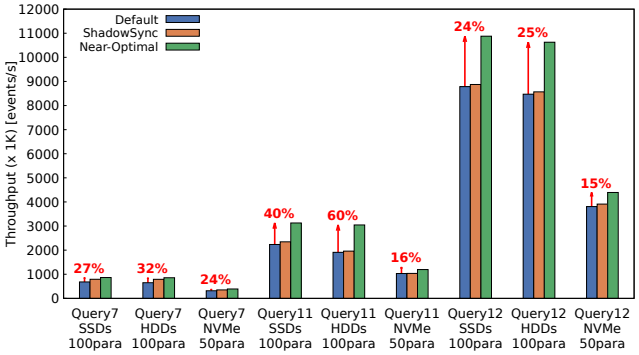


Figure 14: Performance comparison of Queries q7, q11, and q12 under near-optimal soft-resource allocation using the proposed mitigation technique across different disk types, evaluated against the state-of-the-art ShadowSync.

and compaction operations from overlapping, addressing millibottlenecks and reducing tail-latency. As shown in Figure 14, the performance difference between the default configuration and the ShadowSync inspired approach is marginal. This indicates that merely eliminating flush-compaction overlap is insufficient; achieving meaningful performance gains requires careful and near-optimal allocation of flush and compaction threads to better utilize system resources and minimize background I/O interference.

The evaluation of all queries began by identifying the near-optimal stage degree of parallelism for each workload. We first established a baseline by selecting the stage degree of parallelism configuration that achieved the highest throughput under Flink's default configuration. Using this baseline, we compared three RocksDB configurations: the default configuration, a ShadowSync inspired mitigation technique that mitigates flush-compaction overlap, and our near-optimized configuration. While the ShadowSync based approach is effective for compaction-intensive workloads, flush-induced stalls persist and continue to limit stream processing performance. In contrast, our near-optimized configuration further improves throughput by carefully tuning the number of RocksDB flush and compaction threads to better manage background I/O activity. Since RocksDB serves as Flink's default embedded state backend, inefficiencies in these background operations can significantly impact overall streaming performance. Figure 14 presents the performance comparison across the default, ShadowSync inspired, and optimized configurations for different storage backends, demonstrating the effectiveness of our mitigation strategy.

While the relative performance gains on NVMe are less pronounced compared to those observed with HDDs or SATA SSDs, they remain significant. Although NVMe drastically reduces raw I/O latency, it shifts the bottleneck to the CPU and memory bandwidth, where RocksDB's internal management threads compete for resources. By optimizing these parameters, we enable RocksDB to fully exploit NVMe capabilities, mitigating overhead from flush synchronization and compaction. This results in reduced latency variance and stabilized throughput, even under heavy workloads.

These results show that the performance impact of background services like flush and compaction can be substantial, even in queries with a single stateful stage instance. The degree of performance improvement varies across different types of storage media due to differences in I/O speed and bandwidth. Additionally, each query has unique computational characteristics, which affect how the scaling degree of stage parallelism influences performance. For example, lower throughput of HDDs makes them more sensitive to thread contention, hence resulting in larger performance gains when thread allocation is near-optimized. These findings demonstrate that carefully tuning the stage degree of parallelism level and soft resource parameters like flush and compaction threads can significantly enhance Flink's performance, regardless of query complexity or the type of disk used for state management.

6 Discussion

Generalizability of FlushSync. Although our experiments focus on Apache Flink with RocksDB, *FlushSync* is not tied to one implementation. Any stateful streaming engine that persists local state through an LSM-tree-based backend can align background flush and compaction work across operators, especially around checkpoint boundaries. Systems such as Apache Storm [38] and Spark [36], when deployed with similar persistent state backends, may therefore encounter the same transient contention. This suggests that *FlushSync* represents a broader class of off-critical-path bottlenecks in stateful stream processing.

Multifaceted Causes of Contention. Our results identify insufficient soft-resource allocation, particularly flush and compaction threads, as a primary trigger of *FlushSync*. Other configuration and runtime factors can amplify it. Small write buffers force frequent memtable flushes and downstream compactions, while larger buffers trade lower flush frequency for higher memory pressure and recovery cost. JVM garbage collection can also coincide with flush-heavy intervals because Flink continuously creates and discards objects during stream execution. Thus, *FlushSync* should be viewed as coordinated pressure from multiple background maintenance activities, rather than the effect of a single RocksDB parameter.

Disaggregated State Backends. Recent stream processing systems, including Flink 2.0 [26], increasingly separate compute and storage to improve scalability and elasticity. Disaggregation changes where state is accessed and maintained, but it does not eliminate LSM-tree maintenance when

systems still rely on RocksDB-like persistence [14]. Flushes and compactions may move to a different component or node, yet their timing can still affect checkpoint completion, storage throughput, and backpressure. Effective mitigation therefore requires visibility into state-backend background activity even in disaggregated designs.

Towards Automated Mitigation. Our offline tuning method shows that soft-resource allocation can mitigate *FlushSync*, but static settings may become stale as workloads, checkpoint intervals, and state sizes change. A practical next step is a lightweight controller that monitors throughput drops, latency spikes, flush or compaction queuing, and checkpoint duration, then adjusts thread pools and buffer sizes within safe bounds. Such control would complement operator autoscaling by addressing state-backend contention that stage-level metrics alone do not expose.

7 Related Work

Autoscaling in Event-driven Streaming Processing Systems. Previous research on autoscaling primarily targets operator degree of parallelism based on "foreground" business logic load. Standard approaches, particularly in cloud-native environments, scale resources reactively when utilization thresholds (e.g., CPU or throughput) are breached [16, 19, 20, 30, 45, 46]. Similarly, distributed data pipelines often rely on coarse-grained usage patterns for scaling decisions [18, 21, 23, 43]. However, these strategies treat the state backend as a black box; they overlook the cost of asynchronous background services—such as checkpointing, RocksDB flushes, and compactions. In contrast, our work utilizes fine-grained (50 ms) profiling to demonstrate that ignoring these internal mechanisms renders standard autoscalers ineffective, particularly when CPU stalls are induced by *FlushSync*.

Optimizing State Management and Checkpointing. Substantial effort has been dedicated to mitigating the latency overhead of stateful dataflows. Early work focused on optimizing the general efficiency of LSM-tree stores like RocksDB for SSDs [12–14], while more recent studies address specific performance pathologies like write stalls and compaction overhead [2, 5, 24, 31, 42]. Notably, SILK [6] introduces an I/O scheduler to reduce interference between client writes and background operations (flushes/compactions). However, SILK primarily targets tail latency in standalone key-value stores. Our work specifically identifies how these background operations synchronize across a distributed event-driven streaming dataflow to stall the processing pipeline, a phenomenon distinct from local I/O interference.

Software Reconfiguration. Dynamic reconfiguration to mitigate Service Level Objective (SLO) violations in cloud applications is a well-established field [7, 22, 34, 41]. For instance, μTune [34] dynamically selects load-optimal threading models for microservices to minimize tail latency. Similarly, ConScale [22] employs a statistical Scatter-Concurrency-Throughput model to rapidly adapt the concurrency limits of key servers during scaling events. While effective in their respective domains, these approaches have not yet been applied to the internal thread management of stateful event-driven streaming backends. We complement this limitation by introducing a workload-aware tuning mechanism specifically designed to mitigate the interference between background state management and the critical processing path.

Summary. To the best of our knowledge, prior optimizations in event-driven streaming engines have primarily targeted performance variance along the critical processing path. We depart from this by showing that *FlushSync*-induced CPU stalls, originating off the critical path, are a primary driver of throughput degradation. This work is the first to systematically analyze and mitigate these synchronization events in event-driven stream processing systems.

8 Conclusion

Stateful event-driven stream processing engines like Apache Flink have become the de facto standard for continuous, large-scale data analysis. In this paper, however, we demonstrated that the asynchronous state maintenance path, rather than stage degree of parallelism or storage I/O limitations, constitutes the dominant performance bottleneck. We identified and characterized *FlushSync*, a severe pathology where synchronized background operations within RocksDB create transient resource contention that stalls Flink's execution pipeline, even on high-speed NVMe storage. To address this, we proposed an offline tuning methodology for soft resources (e.g., flush and compaction threads) that consistently outperforms naive horizontal scaling, delivering stable throughput and predictable latency. Our findings establish that achieving reliable performance at scale necessitates the proactive management of internal state backend dynamics, rather than merely relying on reactive hardware provisioning. Future work will extend these insights toward lightweight online adaptation mechanisms and cross-engine generalization.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their feedback on improving this paper. This research has been partially funded by the National Science Foundation CNS (2245827) program. Any opinions, findings, and conclusions are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] 2026. *P99 CONF: the event for developers who care about high-performance, low-latency applications.* Retrieved Feb 14, 2026 from https://www.p99conf.io/

[2] Muhammad Yousuf Ahmad and Bettina Kemme. 2015. Compaction management in distributed key-value datastores. *Proceedings of the VLDB Endowment* 8, 8 (2015), 850–861.

[3] Apache Flink. 2026. Apache Flink Metrics Documentation. Retrieved May 1, 2026 from https://nightlies.apache.org/flink/flink-docs-stable/docs/ops/metrics/

[4] Apache Flink. 2026. *Datagen Source.* Retrieved Feb 14, 2026 from https://nightlies.apache.org/flink/flink-docs-master/docs/connectors/datastream/datagen/

[5] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. {TRIAD}: Creating synergies between memory, disk and log in log structured {Key-Value} stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 363–375.

[6] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, 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.

[7] Subho S Banerjee, Saurabh Jha, Zbigniew Kalbarczyk, and Ravishankar K Iyer. 2021. BayesPerf: minimizing performance monitoring errors using Bayesian statistics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 832–844.

[8] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.

[9] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1718–1729.

[10] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, 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.

[12] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *CIDR*, Vol. 3. 3.

[13] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Evolution of development priorities in key-value stores serving large-scale applications: The {rocksdb} experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 33–49.

[14] Facebook. 2026. *RocksDB.* Retrieved Feb 14, 2026 from https://rocksdb.org/

[15] Facebook. 2026. *Tuning Flushes and Compactions.* Retrieved Feb 14, 2026 from https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide#tuning-flushes-and-compactions

[16] Tom ZJ Fu, Jianbing Ding, Richard TB Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. 2017. DRS: Auto-scaling for real-time stream analytics. *IEEE/ACM Transactions on networking* 25, 6 (2017), 3338–3352.

[17] Moojan Ghafurian, David Reitter, and Frank E Ritter. 2020. Countdown timer speed: A trade-off between delay duration perception and recall. *ACM Transactions on Computer-Human Interaction (TOCHI)* 27, 2 (2020), 1–25.

[18] Alim Ul Gias, Giuliano Casale, and Murray Woodside. 2019. ATOM: Model-driven autoscaling for microservices. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1994–2004.

[19] Albert Jonathan, Abhishek Chandra, and Jon Weissman. 2020. WASP: Wide-area adaptive stream processing. In *Proceedings of the 21st international middleware conference*. 221–235.

[20] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 783–798.

[21] Sobhan Omranian Khorasani, Jan S Rellermeyer, and Dick Epema. 2019. Self-adaptive executors for big data processing. In *Proceedings of the 20th International Middleware Conference*. 176–188.

[22] Jianshu Liu, Shungeng Zhang, Qingyang Wang, and Jinpeng Wei. 2020. Mitigating Large Response Time Fluctuations through Fast Concurrency Adapting in Clouds. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 368–377.

[23] Pinchao Liu, Dilma Da Silva, and Liting Hu. 2021. {DART}: A scalable and adaptive edge stream processing engine. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 239–252.

[24] Chen Luo and Michael J Carey. 2019. On performance stability in LSM-based storage systems (extended version). *arXiv preprint arXiv:1906.09667* (2019).

[25] Chen Luo and Michael J Carey. 2020. LSM-based storage techniques: a survey. *The VLDB Journal* 29, 1 (2020), 393–418.

[26] Yuan Mei, Rui Xia, Zhaoqian Lan, Kaitian Hu, Lei Huang, Paris Carbone, Yanfei Lei, Vasiliki Kalavri, Han Yin, and Feng Wang. 2025. Disaggregated state management in apache flink® 2.0. *Proceedings of the VLDB Endowment* 18, 12 (2025), 4846–4859.

[27] Hamid Nasiri, Saeed Nasehi, and Maziar Goudarzi. 2019. Evaluation of distributed stream processing frameworks for IoT applications in Smart Cities. *Journal of Big Data* 6, 1 (2019), 52.

[28] 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.

[29] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. 2020. {FIRM}: An intelligent fine-grained resource management framework for {SLO-Oriented} microservices. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 805–825.

[30] Chenhao Qu, Rodrigo N Calheiros, and Rajkumar Buyya. 2018. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–33.

[31] 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.

[32] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 217–228.

[33] Won Wook Song, Taegeon Um, Sameh Elnikety, Myeongjae Jeon, and Byung-Gon Chun. 2023. Sponge: Fast reactive scaling for stream processing with serverless frameworks. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 301–314.

[34] Akshitha Sriraman and Thomas F Wenisch. 2018. μTune: Auto-Tuned Threading for {OLDI} Microservices. In *13th {USENIX} Symposium on*

Operating Systems Design and Implementation ({OSDI} 18). 177–194.

[35] "The Apache Software Foundation". 2026. *Apache Flink®: Stateful Computations over Data Streams*. Retrieved Feb 14, 2026 from https://flink.apache.org/

[36] "The Apache Software Foundation". 2026. *Dynamic resource allocation in spark*. Retrieved Feb 14, 2026 from https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation

[37] "The University of Utah". 2026. *CloudLab*. Retrieved Feb 14, 2026 from https://www.cloudlab.us/

[38] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 147–156.

[39] Triki, Selima. 2026. The real-time data revolution. Retrieved Feb 14, 2026 from https://digazu.com/the-real-time-data-revolution/

[40] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2008. Nexmark–a benchmark for queries over data streams (draft). *Technical report* (2008).

[41] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. 2018. Understanding and auto-adjusting performance-sensitive configurations. *ACM SIGPLAN Notices* 53, 2 (2018), 154–168.

[42] Hailu Xu, Pinchao Liu, Susana Cruz-Diaz, Dilma Da Silva, and Liting Hu. 2020. SR3: Customizable recovery for stateful stream processing systems. In *Proceedings of the 21st International Middleware Conference*. 251–264.

[43] Le Xu, Boyang Peng, and Indranil Gupta. 2016. Stela: Enabling stream processing systems to scale-in and scale-out on-demand. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 22–31.

[44] Wei D Xu, Matthew J Burns, Frédéric Cherqui, and Tim D Fletcher. 2021. Enhancing stormwater control measures using real-time control technology: a review. *Urban Water Journal* 18, 2 (2021), 101–114.

[45] Zhe Yang, Phuong Nguyen, Haiming Jin, and Klara Nahrstedt. 2019. MIRAS: Model-based reinforcement learning for microservice resource allocation over scientific workflows. In *2019 IEEE 39th international conference on distributed computing systems (ICDCS)*. IEEE, 122–132.

[46] Liang Zhang, Wenli Zheng, Chao Li, Yao Shen, and Minyi Guo. 2021. Autrascale: an automated and transfer learning solution for streaming system auto-scaling. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 912–921.

[47] Shungeng Zhang, Qingyang Wang, Yasuhiko Kanemasa, Julius Michaelis, Jianshu Liu, and Calton Pu. 2022. ShadowSync: latency long tail caused by hidden synchronization in real-time LSM-tree based stream processing systems. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference*. 281–294.