Cluster Computing (2025) 28:420
https://doi.org/10.1007/s10586-024-05000-6

Towards an asynchronous commit in micro-batch streaming systems with log-structured merge-tree based key-value store

Kyuli Park[1] · Dongjae Lee[1] · Yeonwoo Jeong[1] · Salim Hariri[2] · Sungyong Park[1]

Received: 10 October 2024 / Revised: 11 November 2024 / Accepted: 17 December 2024 / Published online: 31 July 2025

Abstract
Micro-batch streaming systems using Log-Structured Merge-tree based Key-Value Store (LSM-KVS) as state stores often experience high tail latency due to several factors. First, the *commit* task is synchronous, blocking query execution until it is fully completed. During this time, the streaming engine must wait for all associated operations to finish. Additionally, remote checkpointing, which is part of the commit task, increases *compaction* time in the LSM-KVS. This involves reading metadata and state from a remote persistent node for compaction then writing the updated data back, which prolongs the commit latency and degrades overall performance. These delays also postpone subsequent tasks, causing rapid data accumulation from data source and creating a cycle that further extends commit latency, ultimately resulting in long tail latency. To address these issues, we propose MɪSA, a micro-batch streaming system that incorporates *asynchronous commit* and *state preloading* mechanisms in the LSM-KVS based architectures. MɪSA overlaps the time-consuming commit operation with query execution and enhances performance through hierarchical state preloading. We implemented MɪSA in Apache Spark Structured Streaming with LSM-KVS support, a widely-used micro-batch streaming platform. Experimental results show that MɪSA reduces tail latency by up to 13.4× at the 99*th* percentile and boosts average throughput by up to 10.4×.

Keywords Micro-batch streaming system · Log-structured merge-tree · Key-value store

Update 12 September 2025 The original online of this article was revised: Some of the corrections were missed to incorporate in the article, the missing corrections have been now incorporated.

✉ Sungyong Park
parksy@sogang.ac.kr

Kyuli Park
kyuripark@sogang.ac.kr

Dongjae Lee
raradong@sogang.ac.kr

Yeonwoo Jeong
akssus12@sogang.ac.kr

Salim Hariri
hariri@arizona.edu

1 Department of Computer Science and Engineering, Sogang University, 35, Baekbeom-ro, Mapo-gu, Seoul, Republic of Korea

2 Department of Electrical and Computer Engineering, The University of Arizona, Tucson, AZ 85719, USA

1 Introduction

Micro-batch streaming systems aggregate incoming data over short intervals and process it as *micro-batch*es. These systems provide higher throughput than event-driven streaming systems by handling large volumes of data at once. They also simplify the execution of complex stateful operations by defining clear boundaries, which improves processing accuracy. Due to these benefits, micro-batch streaming systems have been widely adopted for data analysis in complex stateful queries, such as online fraud detection [1, 2], product recommendation services [3, 4] and network monitoring [5].

As the complexity of stateful queries increases, the size of intermediate states grows substantially, leading to significant memory management overhead and degrading application performance [6]. Additionally, when the state size exceeds the available memory capacity, programs can terminate due to out-of-memory errors, resulting in a loss of state. To mitigate this issue, modern streaming systems [7–10] often rely on external storage for state management.

Among the available options, Log-Structured Merge-tree [11] based Key-Value Stores (LSM-KVS), such as RocksDB [12], LevelDB [13], and Apache Cassandra [14], are widely used for their ability to efficiently store and retrieve large volumes of state information using high-velocity storage devices. LSM-KVS employs append-only writes and in-memory access to reduce the performance impact of disk I/O during query execution. Data is continually written to an in-memory structure called *memtable*, which is later flushed to disk as files. These files are then processed through a background process called *compaction* that reorganizes the data and removes redundancies, optimizing access latency for improved performance.

An LSM-KVS based micro-batch streaming system performs a *commit* task to persist runtime-generated state in LSM-KVS and create checkpoint files for the persistent node. This commit task includes a flushing operation within LSM-KVS and two types of checkpointing operations: LSM checkpointing for the key-value store and stream checkpointing for the streaming engine. When a commit task is initiated from a streaming engine, LSM-KVS operations such as flushing and LSM checkpointing are executed to transfer in-memory states to disk. After these LSM-KVS operations are completed, the streaming engine performs stream checkpointing to transfer the state information from disk to the persistent node. To ensure *exactly-once* semantics during micro-batch processing, all operations within the commit task are executed synchronously. Consequently, even background tasks like flushing in LSM-KVS must wait for preceding commit tasks to finish, which can introduce delays.

The primary delay in commit tasks is due to compaction operations, where longer compaction times result in extended application blocking, which in turn delays subsequent tasks. If a compaction operation is underway before LSM checkpointing begins, the LSM checkpointing must wait for the compaction to finish to prevent data modifications during the checkpointing process. The data accumulated during this delay is transferred to the next batch, causing an unintended increase in both batch size and execution time. Additionally, remote stream checkpointing further extends the commit latency. For instance, a compaction operation requires fetching the necessary metadata from the persistent node and then writing the updated file back. This process increases the overall delay, especially when the persistent node is located remotely. In this paper, commit latency is the time to complete all synchronous operations within a commit task, including flushing, LSM and stream checkpointing, and compaction delays.

To address the aforementioned problems, existing studies [15, 16] have suggested dynamic batch size adjustment mechanisms based on the processing engine's capacity. While these studies can initially reduce latency, they do not fully address the challenge of increasing state size due to long-term stateful computations. By concentrating solely on batch control, these methods fail to resolve underlying issues, resulting in an ongoing cycle of problems. Additionally, adjusting the batch size may reduce throughput and is unlikely to offer a comprehensive solution. Databricks [17] introduces asynchronous stream checkpointing, which allows the next micro-batch to proceed before the checkpointing process is finished, thus reducing latency. However, only the stream checkpointing within the commit task is performed asynchronously, while all other operations within the commit remain synchronous. Consequently, the stream checkpointing overhead is eliminated, reducing commit latency, but commit latency still exists due to other operations within the commit task. Furthermore, it increases the risk of query failures if issues occur during asynchronous checkpointing, as it lacks the retry mechanism present in synchronous checkpointing.

This paper presents MiSA, which introduces *asynchronous commit* and *state preloading* mechanisms for LSM-KVS based micro-batch streaming systems. The key idea behind MiSA is to separate the synchronous commit task from the critical execution path to avoid application stalls. When a commit task is initiated, a dedicated commit thread handles the necessary operations while the micro-batch processing can proceed with the next task concurrently. Additionally, MiSA enables compaction to begin immediately after LSM checkpointing is complete, allowing stream checkpointing and compaction to run in parallel. This overlap permits the commit and compaction thread in the LSM-KVS to execute concurrently. MiSA also introduces a state preloading mechanism, which writes stream checkpointing data locally first, followed by a hierarchical write to DFS. A dedicated checkpointing thread copies the locally stored checkpoint file to DFS, allowing this process to run independently of the commit task. This approach accelerates state retrieval for query execution and speeds up the compaction operation.

We implemented MiSA on Apache Spark Structured Streaming [18] (referred to as vanilla Spark), a widely used distributed micro-batch streaming system. Our experiments with various real-world workloads demonstrate that MiSA reduces tail latency by up to 13.4× and enhances average throughput by up to 10.4×.

This paper makes the following contributions.

- We identified an increase in tail latency due to synchronized commit operation in LSM-KVS based micro-batch streaming systems and analyzed that the compaction operation is one of the main causes of this delay.

- We proposed an asynchronous commit mechanism to address the issue of synchronous commit while preserving exactly-once semantics.
- We proposed a hierarchical checkpointing mechanism called state preloading designed to enhance checkpointing performance.
- We implemented MISA on a real streaming system, improving query processing performance without modifying the core system. This preserves the system's original integrity while ensuring broad compatibility. Consequently, the core principles of MISA can be applied to other micro-batch streaming frameworks.

2 Background and related work

2.1 LSM-KVS

LSM-KVS is a key-value store based on the LSM-tree architecture, designed to optimize write operations using an append-only approach. As depicted in Fig. 1, it consists of both memory and disk components. Incoming write requests, represented as key-value pairs, are initially stored in a memory buffer known as a memtable. Once the memtable is full, it is converted into a Sorted String Table (SST) file and flushed to disk in the background, starting at level 0 (L0).

The disk component is organized into multiple levels, each containing a collection of SST files. At L0, these flushed SST files are unsorted. As data moves to higher levels, the number of SST files allowed per level increases. Once this number exceeds a predefined threshold, a background process called compaction is triggered. During compaction, SST files from levels L_n and L_{n+1} are merged and sorted, creating a new SST file that is stored in L_{n+1}. Notably, within any level except L0, key ranges across SST files cannot overlap.

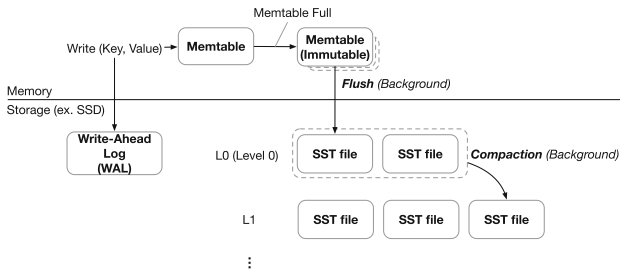
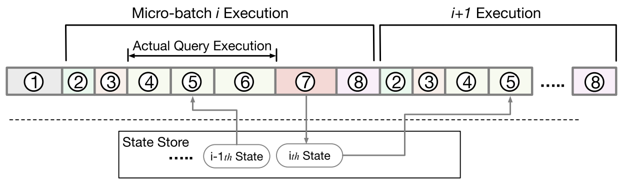


Fig. 1 An overview of LSM-KVS



① Data Ingestion	④ Read Micro-batch Data	⑦ Commit
② Construct Micro-batch	⑤ Read Previous State	⑧ Sink
③ Query Planning	⑥ Query Execution	

Fig. 2 Workflow of micro-batch streaming system

2.2 Micro-batch streaming system

Figure 2 illustrates the workflow of micro-batch streaming systems. In these systems, incoming data is buffered for a specific time interval ①, creating a small set of records called a micro-batch. Once this interval, known as the *trigger* time, expires and the micro-batch is constructed ②, a micro-batch execution begins. The query engine then analyzes the query semantics and data dependencies, generating a query plan represented as a Directed Acyclic Graph (DAG) ③. Afterward, the actual query execution starts, with the plan divided into multiple stages, each containing parallel tasks based on the dependencies between the query's operators.

When execution begins, each partition within a stage, functioning as a parallel execution unit, reads the micro-batch dataset ④. The streaming system then retrieves the previous state generated from the $i-1_{th}$ micro-batch ⑤ and performs query operators such as joins and aggregations ⑥. After completing the query, a commit operation stores the newly generated state to the LSM-KVS ⑦ and commits the intermediate state to a Distributed File System (DFS) [19–21]. Finally, the streaming engine writes the processed results to the output sink ⑧.

2.3 Persistent state management

During micro-batch execution, the streaming engine handles state management by delegating it to the LSM-KVS through commit operations. Once the query execution is completed, the streaming engine commits the state of the current batch to the memtable. Figure 3 illustrates how the state is managed in Spark Structured Streaming using LSM-KVS, providing a detailed view of the process during the commit phase (step ⑦) in Fig. 2.

To ensure that the state is permanently recorded and managed in batch units, the streaming engine explicitly flushes the memtable's contents ①, enabling version control for each micro-batch. In other words, processing a single query results in the creation of an SST file at L0. After flushing, the streaming engine invokes LSM

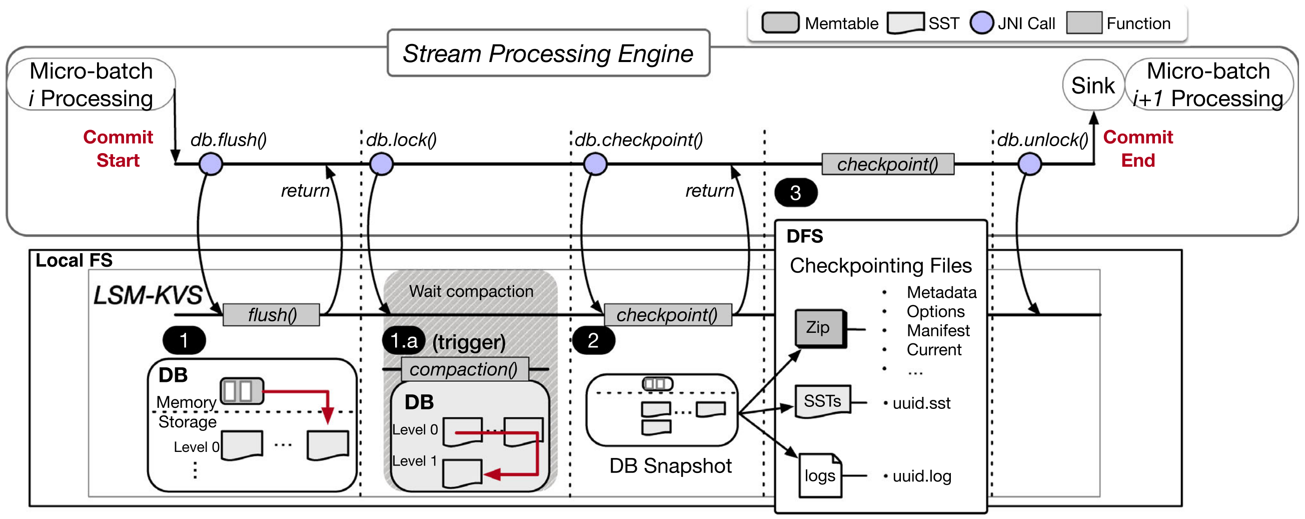


Fig. 3 Persistent state management in Spark Structured Streaming with LSM-KVS. In the LSM-KVS and Stream Processing Engine boxes, checkpoint() refers to LSM checkpointing (db.checkpoint()) and stream checkpointing, respectively. Sink operation is the last step of the Micro-bath *i* Processing

checkpointing function to persist the state ②. However, if a compaction is in progress when the LSM checkpointing is triggered, LSM checkpointing is delayed until the compaction completes ❶ⓐ. Since the checkpointing process creates a snapshot of the LSM-KVS, capturing a consistent state during compaction is difficult due to ongoing changes and updates to data files. As a result, longer compaction times lead to extended wait times for checkpointing, potentially affecting the application performance.

Once the LSM checkpointing starts, LSM-KVS generates several files in its local checkpointing directory, with the most important being the SST file, which contains the actual log-structured data. The remaining files store metadata needed by LSM-KVS to read the SST file and resume from the checkpoint. Since SST files are hard links to files in the LSM-KVS working directory, successive checkpoints may share some of these files. Therefore, these SST files must be copied to a Distributed File System (DFS) in a shared directory. To facilitate this, the streaming engine calls its own stream checkpointing function to persist committed versions ③. It synchronizes the files in LSM-KVS's local checkpointing directory with the shared file system.

Each query execution creates a new stream checkpointing version, storing SST and log files in a designated shared directory for reuse in subsequent queries. Each SST file version is assigned a unique name, allowing the system to identify and read the file using metadata. This enables the next batch to retrieve the previous stream checkpointing version from the file system, recover the state, and continue query execution. The commit task is finalized only after stream checkpointing is completed, at which point the result is written to the sink, and the next micro-batch begins execution.

2.4 Related work

Micro-batch stateful streaming systems Micro-batch processing is a widely used approach in stateful streaming systems. In this method, incoming data is divided into small batches that are processed sequentially, simplifying state management and checkpointing.

Lee et al. introduced zStream [16], which achieves low latency by utilizing dynamic buffering in micro-batch processing based on a reference value known as the *deadline*. zStream reduces tail latency by adjusting the size of each micro-batch dynamically, aiming to complete execution as close to the deadline as possible. A-scheduler [15] proposed an adaptive scheduling mechanism for multiple queries, combined with dynamic micro-batch size control to adapt to traffic fluctuations. This approach achieves low latency by preventing unchecked growth in batch size. However, these two approaches indirectly reduce the number of states by managing micro-batches rather than directly addressing state management.

Databricks [17] provides APIs for managing streaming data and handling incremental changes using Apache Spark Structured Streaming. It supports asynchronous stream checkpointing, which reduces latency by allowing the next micro-batch to proceed without waiting for checkpointing to finish. However, this approach only performs stream checkpointing asynchronously, leaving other operations

Table 1 Summary of previous works on streaming system optimizations

Previous work	LSM-KVS support	Streaming architecture	Optimization metric	Optimization approach	State checkpoint
Meces [22]	✗	Event-driven	Latency	State migration	✔
Rhino [23]	✔	Event-driven	Latency	State migration	✔
FlowKV [24]	✔	Event-driven	Throughput	Customizing KV store	✗
ShadowSync [25]	✔	Event-driven	Latency	Scheduling operation	✔
zStream [16]	✔	Micro-batch	Latency	Dynamic buffering	✗
DataBricks [17]	✔	Micro-batch	Latency	Asynchronous stream checkpointing	✔
MiSA(Ours)	✔	Micro-batch	Latency and throughput	Asynchronous commit	✔

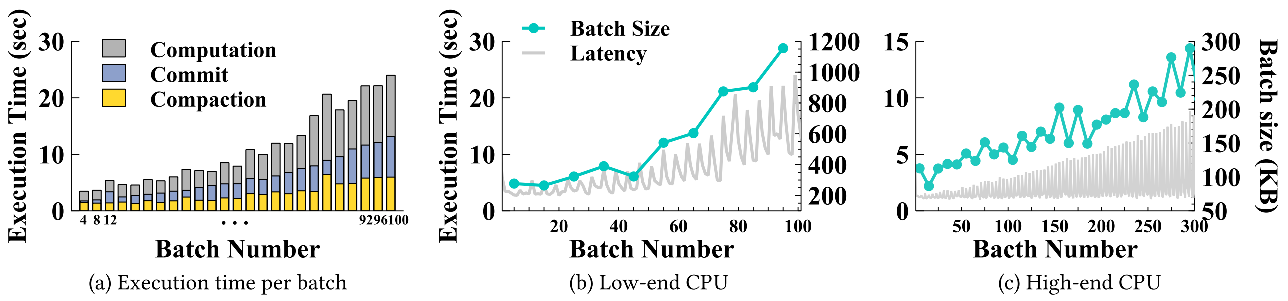


Fig. 4 **a** Execution times per batch when compaction is triggered. In **a**, *Computation* represents the micro-batch processing time, and the total time for each bar is the same as the latency in **b**. **b–c** Correlation between execution time and batch size at various CPU frequencies

within the commit task on the critical path. Consequently, it only partially reduces the duration of the commit task, leading to delays in subsequent operations. Additionally, Databricks acknowledges the limitations of this approach. Specifically, the entire query will fail if a failure occurs during asynchronous stream checkpointing, as it lacks the retry mechanism available in synchronous checkpointing. Furthermore, Databricks is not open-source, which may limit its accessibility and flexibility for certain users.

Event-driven stateful streaming systems Event-driven processing is another approach where each incoming event is processed immediately, allowing for lower latency and higher responsiveness.

Rhino [23] attempted to minimize the amount of state migrated during runtime reconfiguration of large operator states by using a proactive migration protocol while ensuring fault tolerance. It asynchronously replicates the state of a running stateful operator, avoiding any pause in query execution. Meces [22] aimed to reduce latency through state migration, enabling prioritized state transfers during rescaling to ensure that the most critical states are migrated first. This is achieved without additional resource usage during non-rescaling periods. However, it does not support LSM-KVS, which limits its applicability in certain scenarios. Zhang et al. [25] identified latency spikes in stream processing engines using LSM-KVS when LSM operations overlap. To address this, they proposed a scheduling strategy to minimize overlapping LSM operations. Meanwhile, Lee et al. introduced FlowKV [24], a specialized LSM-KVS optimized for window operations, highlighting the inefficiencies in the interaction between stream processing applications and state backends.

Checkpointing Checkpointing is a well-known method used to achieve fault tolerance in distributed systems. In this approach, the system periodically stores the application state to persistent storage, enabling recovery in the event of a failure.

Sachini et al. [26] proposed a mathematical model for multi-level checkpointing based on utilization in stream processing systems. They define utilization as the fraction of time spent processing data and overhead as the fraction of time spent recovering from failures. Using these definitions, they identify the optimal value that maximizes utilization. Chiron [27] determines the optimal checkpointing frequency for distributed stream processing applications by balancing the trade-off between end-to-end latency and total recovery time in case of failure.

Compaction optimization Although widely used in latency-sensitive applications, LSM-tree-based key-value stores incur significant overhead due to their internal operations, particularly compaction, which is resource-intensive. Several studies, such as [28, 29], have aimed to reduce this overhead by optimizing and scheduling compaction. While these optimizations help decrease the commit times, they cannot entirely eliminate delays in subsequent tasks caused by blocking applications. To address this, we remove the commit task from the critical path, avoiding the endless cycles caused by compaction.

Table 1 provides a summary of previous studies on optimizations in streaming systems.

3 Motivations

To investigate the impact of operations during the commit task, we evaluated the performance of vanilla Spark using the Linear Road benchmark [30], a real-world road traffic monitoring benchmark built on the streaming pipeline, with RocksDB [12] as the state store. The detailed system specifications for each server and workloads are outlined in Sect. 5.1. The experiments utilized LR2 queries (shown in Table 2), with the trigger interval in vanilla Spark set to the default value of 1 s and a traffic rate of 1000 records per second. The size of each record was 65 bytes.

3.1 Problem 1: synchronous commit

As explained in Sect. 2.3, the flush operation during a commit task generates an SST file in the LSM-KVS. When the number of SST files exceeds a predefined threshold, a compaction operation may be triggered. We set this threshold to 4, ensuring compaction is triggered after every 4*th* batch.

Figure 4a presents the execution times of commit and compaction operations over the first 100 batches, illustrating when compaction takes place. As depicted in the figure, the steady increase in commit time leads to a substantial rise in overall execution time. This happens because the commit operation is executed synchronously, causing delays in subsequent tasks until the commit finishes. When compaction takes place, the next operation in the commit task (e.g., LSM checkpointing) is forced to wait, which increases the size of the micro-batch. This creates a cycle where the growing state size leads to longer commit and compaction times, further exacerbating delays.

To further explore the correlation between execution time and batch size, we conducted the same benchmark on two servers with different CPU frequencies (i.e., a low-end CPU and a high-end CPU). The purpose of using different CPU frequencies was to demonstrate that, even though compaction may be faster on a high-end CPU, the commit time remains prolonged, eventually leading to an increase in batch size. Figure 4b, c show the execution time and accumulated micro-batch size based on CPU specifications. In this experiment, the micro-batch size was calculated by sampling and averaging 10 batches. Although the patterns of growth in execution time and batch size differ,[1] both metrics ultimately exhibit increases regardless of the CPU.

As depicted in Fig. 4(b), the server with the low-end CPU experiences a sharp increase in execution time. For example, by the 60*th* batch, latency reaches approximately 7.6 s, representing a $2\times$ increase from the initial batch phase. By the 100*th* batch, latency rises to nearly 17 s, a $5.3\times$ increase from the 10*th* batch. Additionally, the batch size processed between the 90*th* and 100*th* batch is $4.17\times$ larger than in the initial batch phase. A similar trend is observed for the server with the high-end CPU, as shown in Fig. 4(c). While the higher-end CPU reduces compaction time, slight delays still occur, causing a gradual accumulation in batch size.

In summary, LSM-KVS based micro-batch streaming systems running for a long period suffer from significant delays, regardless of CPU specifications. A key factor contributing to these delays is the synchronous nature of the commit operation.

3.2 Problem 2: overhead in remote checkpointing

In High Performance Computing (HPC) applications and streaming systems, remote checkpointing is often carried out on remote machines to ensure fault tolerance [31–34]. This study aims to quantify the overhead incurred when stream checkpointing is performed on remote nodes.

Figure 5 illustrates the latency of four key operations—*Get*, *Compaction*, *L.Ckp* (LSM checkpointing), and *S.Ckp* (stream checkpointing))—which affect the 99*th* percentile tail latency when checkpointing is performed either locally or remotely. In both cases, checkpointing was conducted on worker nodes with low-end CPUs. The x-axis represents the checkpointing location (local or remote), while the y-axis uses a different scale for clarity.

As shown in Fig. 5, checkpointing on a remote node results in a noticeable increase in latency for compaction and stream checkpointing operations. Compaction on the remote node takes 7 s, approximately $6.7\times$ longer than on the local node. Likewise, stream checkpointing on the remote node has a latency of 2.5 s on the remote node, a 36.8-fold increase compared to local checkpointing. This increase is primarily due to network overhead, as the checkpointing data must be stored in the remote DFS. Additionally, compaction is further delayed because files need to be transferred between local and remote nodes, requiring read, process, and send operations across locations. The get operation also experiences latency when executed remotely since the state data must be fetched

[1] Note that the y-axis in both figures uses a different scale.

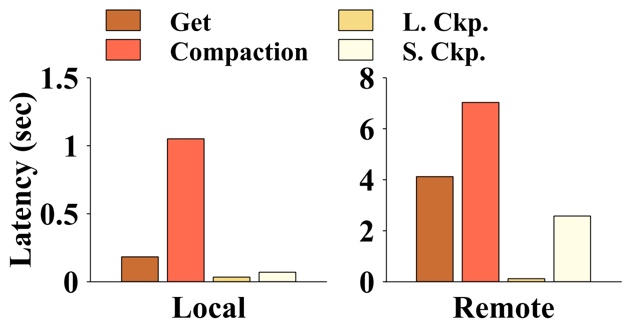


Fig. 5 Operations influencing the 99*th* percentile tail latency depending on the checkpointing location. L.Ckp refers to LSM checkpointing, while S.Ckp represents stream checkpointing. All experiments were performed in a low-end CPU environment

from the remote node during a query. It takes around 4.1 s on the remote node, compared to just 0.2 s locally, making it about 20 times slower. However, the LSM checkpointing latency remains relatively low, 121 ms for remote and 34 ms for local checkpointing, as it is performed on the local node where the LSM-KVS resides. As a result, LSM checkpointing is not impacted by the location of the stream checkpointing, leading to minimal differences between local and remote configurations at the 99*th* percentile tail latency.

In summary, remote checkpointing significantly increases the latency of operations such as get, compaction, and stream checkpointing, which degrades tail latency and overall performance. In contrast, local checkpointing shows only slight latency increases. This analysis confirms that remote checkpointing introduces considerable latency overhead, negatively affecting system performance.

4 Design and implementation

4.1 Overview of MiSA architecture

We propose MiSA, a micro-batch streaming system based on LSM-KVS that integrates asynchronous commit and state preloading mechanisms. MiSA aims to achieve two key objectives: (i) reducing tail latency by overlapping query processing with asynchronous commit tasks. (ii) Enhancing access to persistent data during query execution and minimizing stream checkpointing time through the state preloading mechanism. MiSA is implemented on the JAVA-based micro-batch streaming framework, Apache Spark Structured Streaming, using RocksDB as its state store.

Figure 6 presents the architecture of MiSA, which comprises two main components: Streaming Processor and MiSA Controller. Streaming Processor manages query processing and initiates commit tasks as threads, while MiSA Controller executes these tasks asynchronously.

The flow of MiSA operates as follows. Once the query processing for the i_{th} micro-batch is completed, Streaming Processor sends a signal to MiSA Controller. At the same time, the results are written to the sink, and the system immediately begins the $i+1_{th}$ micro-batch. Upon receiving the signal, MiSA Controller initiates the commit asynchronously via the commit thread ①. The contents of the memtable are then flushed to the LSM-KVS ②, with the flushing operation explicitly triggered by calling the flush function through the Java Native Interface (JNI).

Once the flushing operation is completed, the commit thread calls the JNI function to perform LSM checkpointing, capturing a snapshot of the LSM-KVS ③. If compaction is triggered within LSM-KVS at this time, it proceeds independently via a dedicated compaction thread. However, running compaction and LSM checkpointing simultaneously can compromise data consistency. Therefore, MiSA ensures that LSM checkpointing and compaction do not occur concurrently by locking the LSM-KVS during checkpointing.

Regardless of compaction, once the LSM checkpointing is completed, the commit thread proceeds with the stream checkpointing ④ on the local machine, followed by sending an acknowledgment signal (Ack) to the Streaming Processor ⑤. Meanwhile, the commit thread creates a stream checkpointing thread to copy the checkpoint files to the remote DFS asynchronously ⑥. Finally, MiSA Controller sends an Ack to the currently executing $i+1_{th}$ micro-batch ⑦, indicating that the commit is complete.

To simplify the explanation of the MiSA architecture, Algorithm 1 provides a pseudocode that outlines the asynchronous commit workflow of MiSA. Details on how MiSA overlaps with other stream processing and the state preloading mechanism are covered in Sects. 4.2 and 4.3.

4.2 Overlapping flow

Figure 7 illustrates how MiSA overlaps with micro-batch processing to improve its performance. The overlapping scenarios are categorized into three cases. Case I represents a scenario with no compaction, while Case II and Case III involve compaction being triggered. In all cases, the flow prior to commit remains the same.

As stream processing starts, data is continuously ingested and organized into micro-batches. A DAG is then created for query planning. Once the query is ready for execution, the micro-batch data is processed alongside the previously generated state to perform the actual query (①–⑥). Up to this point, the process is identical to that shown in Fig. 2 of Sect. 2.2.

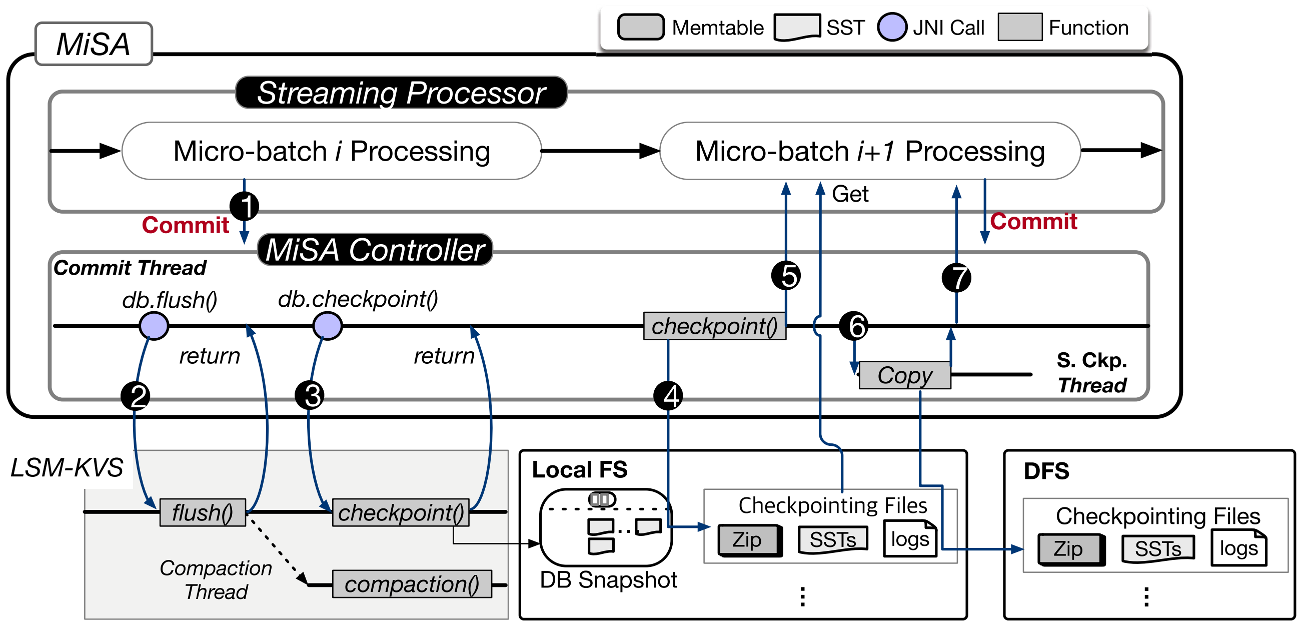


Fig. 6 An overview of MiSA architecture. S. Ckp. denotes stream checkpointing. In the LSM-KVS and MiSA Controller boxes, checkpoint() refers to LSM checkpointing (db.checkpoint()) and stream checkpointing, respectively

After the query execution finishes, the commit task begins asynchronously ⑦ while the sink operation for the micro-batch continues ⑧. Following this, the next micro-batch processing starts. During the commit task, the flushing, LSM checkpointing, and stream checkpointing operations are executed sequentially (⑨ ∼ ⑪). Since these steps are independent, they must be completed in order. This process represents Case I in Fig. 7, where commit tasks and micro-batch processing can overlap for improved performance.

If compaction occurs, two possible scenarios exist depending on when it is triggered. In Case II, compaction is triggered before LSM checkpointing. Here, compaction must be completed first, followed by LSM checkpointing. Thus, the commit task proceeds in the order of ⑨, ⑫, ⑩, and ⑪. During this process, subsequent tasks may be delayed due to the longer time required for compaction, potentially pushing the start of ④ beyond its intended time. Consequently, ⑤ may have to wait until the entire commit task is finished. In Case III, compaction is triggered after LSM checkpointing. In this case, LSM checkpointing can proceed immediately without waiting for compaction, reducing delays.

Stream checkpointing and compaction have no dependencies between each other. Once the target file is flushed and the LSM checkpointing is completed, compaction can proceed using a copy of the file, while stream checkpointing continues simultaneously, as it only depends on the completion of LSM checkpointing. To take advantage of this, MiSA enables the commit thread of the MiSA Controller to overlap with the compaction thread managed within LSM-KVS. This allows the two operations not only to overlap with each other but also with ongoing micro-batch processing.

Typically, compaction takes longer than stream checkpointing but shorter than micro-batch processing. This arrangement ideally allows commit tasks to be completely hidden, enabling each micro-batch to be processed immediately after the previous one without any delays.

By isolating the commit task from micro-batch processing and separating the compaction process within the commit, the time spent on commits does not count toward the micro-batch processing duration. The overlapping portion of the commit occurs between the steps of writing the results to the sink and reading the micro-batch data during the actual query execution of the next micro-batch.

4.3 State preloading

In traditional systems, stream checkpointing is executed directly on a remote node, leading to remote I/O overhead for every state access. When stream checkpointing is performed directly on DFS to save a file, micro-batch streaming is executed using the state stored at this remote location. This approach introduces additional overhead, not only from the stream checkpointing process itself but also from compaction and state fetching (i.e., get operations) required for query execution, leading to increased latency.

In contrast, The state preloading mechanism proposed in MiSA minimizes network overhead involved in all state processing by directly performing stream checkpointing locally. This technique initially writes checkpoint data

Algorithm 1 MiSA

```
Input: $i$ - Micro-batch execution number
Function MicrobatchExec($i$):
    // Construct and read the $i_{th}$ micro-batch
    Construct a microbatch and read the contents;
    // Read the $(i-1)_{th}$ state
    while read state Ack from the $(i-1)_{th}$ commit is not received do
        ;
    Read previous state from the $(i-1)_{th}$ commit;
    // Execute query
    Execute the $i_{th}$ query;
    // Start a thread for commit task
    while two commit Acks from the $(i-1)_{th}$ commit are not received do
        ;
    Create a Commit thread with state information;
    // Write the processed results to the output sink
    Sink the results;
    Return;
Input: $state$ - state information
Thread Commit($state$):
    // Flush
    Flush the state using JNI call;
    // LSM checkpointing
    while compaction is in progress do
        ;
    Activate LSM checkpoiting operations using JNI call;
    // Stream checkpointing on local node
    Stream checkpointing using LSM checkpointing result;
    Send a read state ACK to MicrobatchExec;
    // Start checkpoint thread
    Create a Checkpoint thread with checkpoint file;
    if compaction is completed then
        Send a commit ACK to MicrobatchExec;
Input: $file$ - checkpoint files
Thread Checkpoint($file$):
    copy file to DFS;
    Send a commit Ack to MicrobatchExec;
```

locally and then hierarchically transfers files to DFS using the stream checkpoint thread. Transferring the checkpointing file to DFS is to safeguard against data loss due to failure. MiSA Controller accomplishes this by modifying the read path for these operations from remote storage to local storage through orchestration between the stream checkpoint thread and the commit thread.

MiSA Controller ensures that stream checkpointing during a commit occurs on the local file system. It generates the current version of the checkpoint file using the snapshot created during LSM checkpointing. Once stream checkpointing is finished, it sends the first Ack to Streaming Processor. Subsequently, the checkpoint files are transferred to the remote file system through a stream checkpoint thread. After the transfer to the DFS is complete, the stream checkpoint thread sends an Ack to the commit thread. Upon receiving this Ack, MiSA Controller then sends the second Ack to Streaming Processor.

Streaming Processor receives two Acks for the following reasons: the first Ack is required to read the state for the next micro-batch, while the second Ack is necessary to trigger the next commit, thus enabling its execution. In the event of a failure during local stream checkpointing that results in data loss, the most recent stream checkpoint version can be retrieved from the remote node, ensuring fault tolerance.

4.4 Keeping consistency

Asynchronous commit can introduce consistency issues. Figure 8 shows three scenarios that uphold the execution order of the system to prevent these consistency problems.

Figure 8a illustrates a scenario where micro-batch processing and commit tasks can overlap. As explained earlier, the commit occurs asynchronously after the execution of the i_{th} query. The sequence of operations within the commit includes flushing, LSM checkpointing, and stream checkpointing. All three of these operations must be completed before the state reading operation for the next $i+1_{th}$ micro-batch processing. Here, stream checkpointing specially refers to the checkpointing performed to a local directory (i.e., up to the first Ack). If stream checkpointing is not yet finished, the state reading operation will be delayed, waiting for its completion and acknowledgment, thereby ensuring that the correct state generated from the previous processing is accessed.

Figure 8b depicts the required order of operations when compaction occurs. If compaction is initiated after the flushing operation, it will be executed in the background thread of LSM-KVS, allowing it to run concurrently with stream checkpointing. However, compaction must be completed prior to the $i+1_{th}$ commit, which occurs after the execution of the $i+1_{th}$ query. During state retrieval, the SST file from stream checkpointing must be accessed, and if compaction is not finished, this could lead to slower read performance. Therefore, if compaction is not completed before the next commit, the commit task will pause until compaction is finished to ensure faster state reading.

Figure 8c shows the order that must be followed when copying checkpoint files to the remote DFS after stream checkpointing. Once stream checkpointing is performed in the local directory, the checkpoint file is copied to DFS in the background. This operation is required to be completed before the $i+1_{th}$ commit, as depicted in Fig. 8c. While it is not critical for the file to be copied to DFS before the next commit, doing so is essential for state retrieval in the event of a failure. Furthermore, to guarantee that each checkpoint version is stored in DFS, this copy operation must be finalized before the commit associated with the

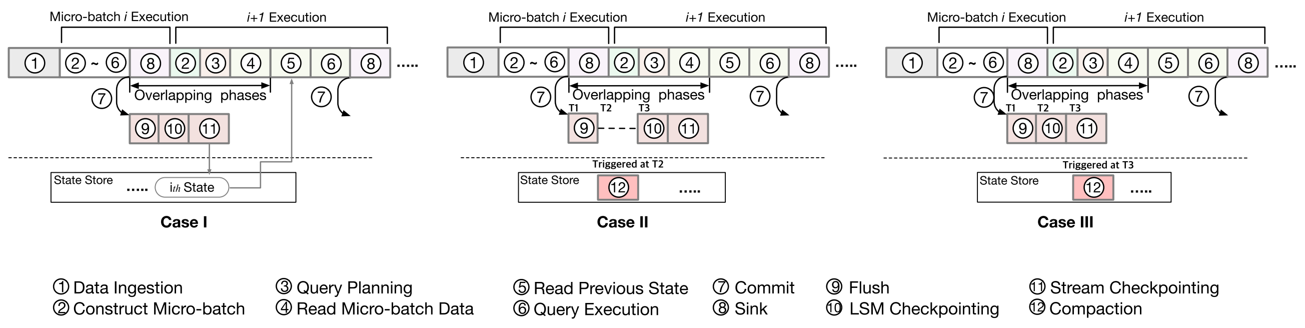


Fig. 7 Three overlapping cases in MiSA. T1, T2, and T3 represent the time axes in Case II and Case III. Case II and III write the i_{th} state to the state store and read the i_{th} state during the $i + 1_{th}$ execution like Case I

stream checkpointing. As shown in Fig. 8c, if the acknowledgment confirming the successful copy to DFS is not received prior to the next commit, the system will pause until the checkpoint file is successfully copied, ensuring the preservation of each checkpoint version.

4.5 Failure handling

Since MiSA performs commits through an asynchronous thread, failure handling is required. The following scenarios may arise during an asynchronous commit: a failure during stream checkpointing and after stream checkpointing. Here, a failure is defined as the termination of the commit thread. In both scenarios, the query may indefinitely await an Ack due to the absence of a response. To prevent this, system status is periodically monitored using a heartbeat mechanism.

In the first scenario, failure occurs during stream checkpointing. If the system does not receive a heartbeat signal, it detects a failure and checks for the presence of the checkpoint file. If the checkpoint file is missing, it indicates that stream checkpointing was not completed successfully. In this case, the commit thread is recreated, and the commit task is restarted.

In the second scenario, failure occurs after stream checkpointing. The checkpoint file, that is state, has been written locally, but the commit thread terminates before sending an ack. If the heartbeat is not received, the system detects a failure and checks for the checkpoint file. Since the checkpoint file exists, the state is read and used to proceed with query execution as usual.

Additionally, query execution is represented as a DAG, with multiple operators arranged in sequence. In a DAG-based execution model, operators are processed sequentially on worker nodes, while the master node continuously tracks the completion status of each operator. This tracking allows the system to recover from intermediate failures by restarting from the failed operator, as the master node retains the success history of prior operators. This failure recovery model is a default mechanism in DAG-based streaming systems with a master/worker architecture. As MiSA follows this model, it can resume processing from the failure point if a worker node fails, ensuring efficient fault tolerance and minimizing reprocessing overhead.

4.6 Implementation

We implemented MiSA on Apache Spark Structured Streaming (version 3.2.3) [18] framework. Specifically, we focused on one of the native core modules, Spark-SQL [36], which performs various operations during the commit task and while executing queries. The overall system configuration was implemented using Scala language [37], and we utilized *Future* [38], a standard Scala library that supports asynchronous programming.

MiSA's design aims to make the commit process (step ⑦ in Fig. 2) asynchronous, enabling commits to overlap with other tasks, and to asynchronously copy checkpoint data to DFS. As a result, implementing these concepts in existing systems is straightforward. MiSA achieves this by creating a separate thread each time the original commit function is called, allowing for asynchronous operation. Since transitioning to asynchronous processing may disrupt the original synchronous sequence, acknowledgments (acks) are used to ensure synchronization. Additionally, the stream checkpointing path is modified to be local, with a thread that asynchronously copies data to DFS and generates an ack to synchronize with the main process. MiSA also leverages built-in LSM-KVS functions to monitor background tasks (e.g., compaction) within LSM-KVS. These adjustments make integrating MiSA into existing systems easier.

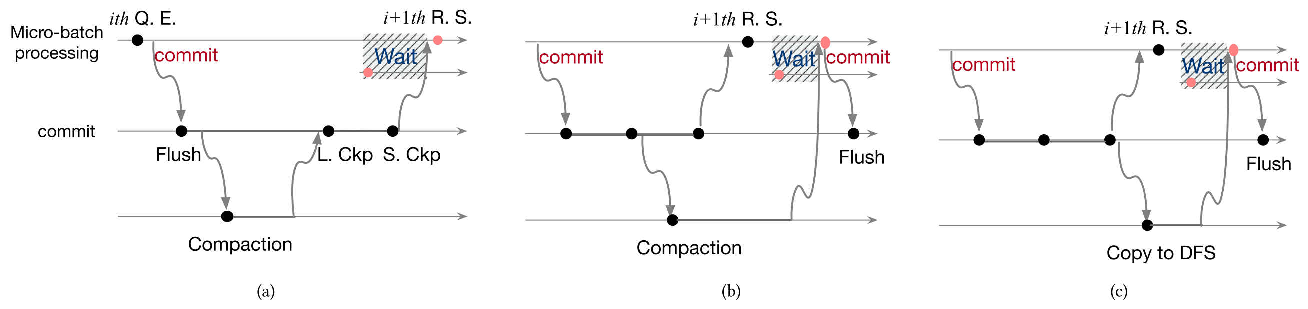


Fig. 8 **a**, **b** and **c** represent three scenarios to ensure consistency. Q.E. and R.S. denote Query Execution and Read State, respectively. L.Ckp and S.Ckp denote LSM checkpoint and Stream checkpoint, respectively. The order of Flush, L.Ckp, and S.Ckp in commit is the same in both **b** and **c**. Note that the micro-batch processing and commit arrows show sequences, not threads

Table 2 Query details of real-world streaming workloads used in experiments

Benchmark	Notation	Query details
Linear road [30]	*LR*2	SELECT L.timestamp, L.vehicle, L.speed, L.highway, L.lane, L.direction, L.segment FROM SegSpeedStr [range 30 (slide 1)] as A, SegSpeedStr as L WHERE (A.vehicle == L.vehicle)
	*LR*4	SELECT timestamp, highway, direction, segment, COUNT(vehicle) as num Vehicle
		FROM SegSpeedStr [range 20 slide 5] GROUPBY (highway, direction, segment)
Cluster monitoring [35]	*CM*1	SELECT timestamp, category, SUM(cpu) as totalCpu
		FROM TaskEvents [range 60 (slide 10)] GROUPBY category ORDERBY SUM(cpu)
	*CM*2	SELECT jobId, AVG(cpu) as avgCpu
		FROM TaskEvents [range 30 slide 1] WHERE (eventType == 1) GROUPBY jobId

5 Evaluation

5.1 Experimental setup

Configuration For the experiments, we set up a Spark cluster with one master node and two worker nodes. Each worker node hosted a Spark executor with 8 CPU cores and 48 GB of memory, and we used 8 data partitions to facilitate parallel processing. To evaluate the impact of different CPU specifications on performance, we configured two Spark clusters: one equipped with high-end CPUs and the other with low-end CPUs. Detailed specifications can be found in Table 3.

Workloads The workloads used in the experiments are outlined in Table 2. We selected four stateful queries from the real-world streaming benchmarks, the Linear Road benchmark [30] and Cluster Monitoring benchmark [35]. LR2 includes a join operator that is both compute-intensive and resource-demanding due to its 1-second sliding window. In contrast, LR4 use aggregate operators to compute count rows, with sliding window set to 5 s to highlight the differences between the queries. Additionally, LR4, having larger sliding windows than LR2, result in lighter workloads Likewise, CM1 has a sort operator, making it compute intensive, whereas CM2 uses aggregate to calculate the average, resulting in lower computational demands than CM1. We also set the trigger value to 3 s to accumulate more records for each batch, as this value controls the duration the streaming engine buffers incoming records before processing.

Input traffic We generated random traffic with varying numbers of records[2] per second (records/sec). The record count follows a normal distribution, with the average converging to the specified traffic rate. Unless otherwise noted, the traffic rate was set to 1500 records/sec, which is adequate to demonstrate the challenges encountered by existing micro-batch streaming systems. To achieve this traffic, we adjusted the input from the Apache Kafka [10] broker to attain an average of *n* records/sec.

Comparison targets To demonstrate the effectiveness of our approach, we selected three comparison targets: vanilla Spark and two optimized versions. We designated vanilla Spark as the baseline (denoted as `Baseline`) since it is one of the most widely used micro-batch streaming systems. This allows us to highlight the issues that arise when using LSM-KVS as a state store. The other two comparison targets include one that utilizes dynamic buffering (denoted

[2] The size of one record is 65 bytes.

as `zStream`) and another that implements asynchronous stream checkpointing (denoted as `Async.Ckp`). Both modified versions of Apache Spark Structured Streaming aim to optimize latency. However, we will show that their methods are insufficient to address the latency issues in micro-batch streaming system that use LSM-KVS. We implemented these modifications using Apache Spark Structured Streaming for comparison, and details are summarized below.

- `Baseline`: LSM-KVS based vanilla Spark.
- `zStream`: LSM-KVS based vanilla Spark with dynamic buffering mechanism [16].
- `Async.Ckp`: an implementation of Databricks' approach in vanilla Spark.

In `zStream` [16], a dynamic buffering mechanism using *deadline* is employed to control the batch size at runtime. The deadline serves as the maximum time allowed for query processing. Specifically, the engine continues to buffer incoming records only as long as the current micro-batch can be processed within this deadline. If the expected processing time for the current micro-batch exceeds this limit, buffering stops, and any remaining input records are carried over to the next batch. Since the concept of a deadline is similar to the trigger value in Apache Spark Structured Streaming (i.e., dynamically adjusting the trigger value), we set it to 3 s for comparison.

`Async.Ckp` aims to improve performance by shifting the stream checkpointing of Apache Spark Structured Streaming, which is usually done synchronously, to the background. This approach, proposed by Databricks [17], shares similarities with our method but has a significant limitation: it only decouples the stream checkpointing process, which is part of the commit task that we identify as problematic. Since Databricks' implementation is not available as open source, we implemented this approach using Apache Spark Structured Streaming to demonstrate its limitations and to show how our solution surpasses it.

5.2 Overall performance

Figures 9 and 10 compare the execution time of each batch in MiSA and its comparison targets across the four queries (LR2, LR4, CM1, CM2) for up to the 100*th* batch. The experiment utilizes two types of traffic rates: 1500 records/sec (normal) and 2500 records/sec (heavy). The x-axis represents the batch number, while the y-axis indicates the execution time for each batch. Throughout the experiment, execution times for MiSA and all comparison targets fluctuate. We evaluate their performance by analyzing the frequency and degree of these fluctuations, as well as the overall trend.

Figure 9a illustrates the execution times of LR2 at a traffic rate of 1500 records/sec. The figure shows that all three comparison targets periodically experience sharp spikes in execution time, with the peaks of these fluctuations rising as the batch number increases. These fluctuations are result from the regular triggering of compaction in LSM-KVS, which delays subsequent tasks in the commit process until compaction is complete. This unintended consequence of using LSM-KVS as a state store emphasizes the importance of carefully considering LSM-KVS operations when optimizing performance.

Notably, `Async.Ckp` exhibits a pattern similar to the other two comparison targets, despite its consideration of LSM-KVS. This suggests that its approach of decoupling only the stream checkpointing, which is part of the commit process, is inadequate for fully optimizing the system. In contrast, while MiSA also experiences fluctuations, its peak values remain relatively stable, and the execution time does not increase over time. Although it cannot completely eliminate compaction time, the frequency and magnitude of these fluctuations are significantly smaller and more stable compared to the other three systems.

When the traffic rate for the same query is 2500 records/sec (as shown in Fig. 10a), the systems behave similarly to when the traffic rate is 1500 records/sec at least until around the 55*th* batch. However, the effects are more pronounced due to the increased volume of incoming data and the computationally intensive join operator. Beyond this point, while MiSA continues to show stable performance, the other comparison targets experience the previously mentioned vicious cycle more severely, leading to delays so significant that further analysis becomes impractical or irrelevant. Therefore, in the subsequent experiments, we will focus on the results up to the 55*th* batch for LR2 at a traffic rate of 2500 records/sec.

The LR4 displays similar patterns under both traffic conditions(Figs. 9b, 10b). Unlike LR2, where all three comparison targets exhibited progressively increasing values, LR4 maintains a consistent peak value. This consistency is due to the lighter workloads of these two queries compared to the more computationally intensive LR2. As a result, the compaction time is shorter, and the overhead from processing additional incoming data is minimal, preventing the vicious cycle seen in LR2.

Table 3 Testbed specification

CPU (high-end)	AMD Ryzen 9 3900X 12-core 3.80 GHz
CPU (low-end)	AMD Ryzen 9 3900X 12-core 2.80 GHz
Memory	DDR4, 64 GB
Storage	Samsung SSD 970 EVO NVMe SSD
Ethernet	1 Gbps

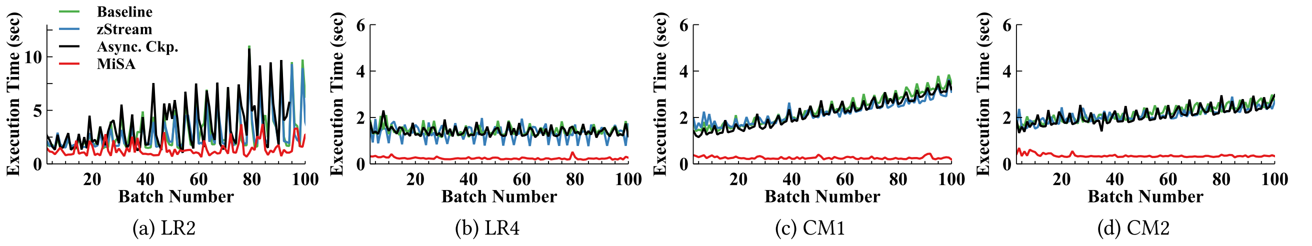


Fig. 9 **a**, **b**, **c**, and **d** correspond to the benchmarks LR2, LR4, CM1, and CM2, which indicate specific queries. Execution times for the four queries (LR2, LR4, CM1, CM2) at a traffic rate of 1500 records/sec on a high-end CPU. Note that figure **a** uses a different scale on the y-axis

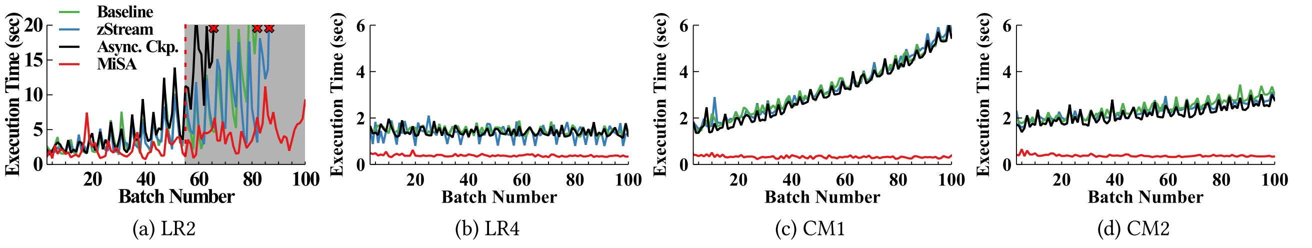


Fig. 10 **a**, **b**, **c**, and **d** correspond to the benchmarks LR2, LR4, CM1, and CM2, which indicate specific queries. Execution times for the four queries (LR2, LR4, CM1, CM2) at a traffic rate of 2500 records/sec on a high-end CPU. In LR2, the red dotted vertical line marks the 55*th* batch. Note that figure **a** uses a different scale on the y-axis

CM1 and CM2 exhibit similar spike patterns, reaching a consistent peak value similar to LR4 under both traffic conditions. However, execution time for both queries shows an overall increasing trend regardless of traffic. In particular, CM1, which is more compute-intensive than CM2, shows a more apparent increase in execution time. It is visible that the slope of the comparison systems becomes steeper at a traffic rate of 2500 (Fig. 10c). Nevertheless, MiSA consistently achieves the best performance by maintaining a stable execution time.

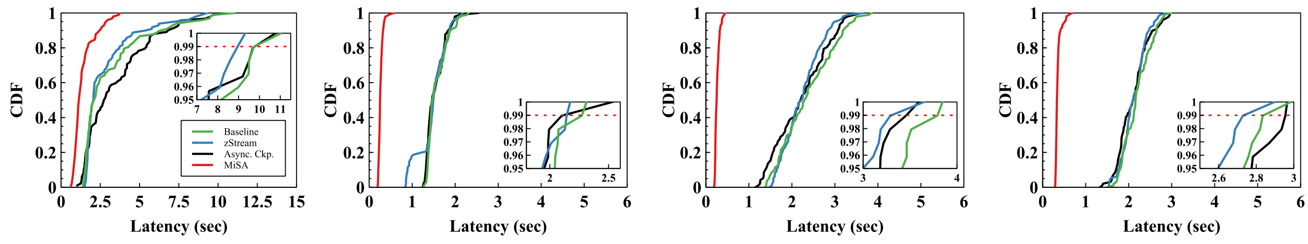
5.3 Tail latency and throughput

Figures 11 and 12 illustrate the Cumulative Distribution Functions (CDFs) of latencies for MiSA and the comparison targets across the four queries at two traffic rates: 1500 records/sec and 2500 records/sec. In these figures, the x-axis represents latency, corresponding to the execution times of batches up to the 100*th* batch, while the y-axis indicates the percentile of the cumulative distribution of these execution times. The 99*th* percentile tail latency is marked with a red dotted line in a inset figure.

Overall, as shown in these figures, MiSA consistently demonstrates lower and more stable latencies across different queries and traffic rates. In particular, the 99*th* percentile tail latency is significantly lower than those of the comparison targets. For instance, in LR2 with a traffic rate of 1500 records/sec, as shown in Fig. 11a, the latency for MiSA starts increasing around the 80*th* percentile, whereas the comparison targets experience a sharp rise between the 35*th* and 65*th* percentiles. At the 99*th* percentile, MiSA achieves approximately a 2.6× speedup over the worst-performing `Baseline` (3694 ms vs. 9729 ms).

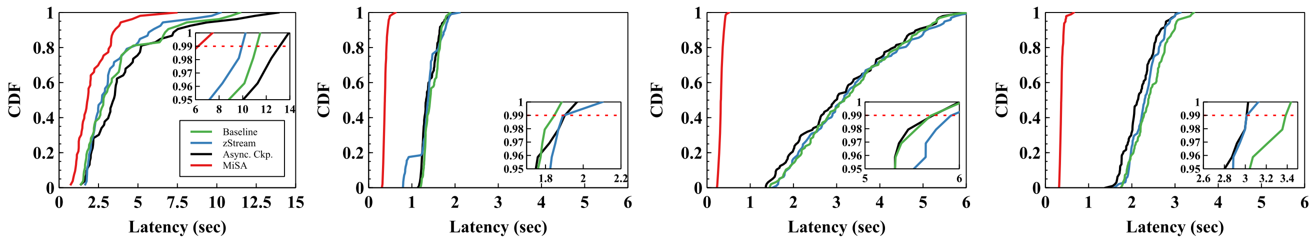
It is notable that MiSA outperforms `zStream` in terms of the 99*th* tail latency, which was specifically designed to reduce tail latency. We believe this is because MiSA achieves substantial performance gains by eliminating unnecessary synchronization between the internal operations of the LSM-KVS and the streaming engine. In contrast, `zStream` focuses solely on optimizing the streaming engine without addressing the LSM-KVS, limiting its ability to achieve maximum optimization when using LSM-KVS as the state store.

When the traffic rate increases to 2500 records/sec in LR2, a similar pattern is observed as shown in Fig. 12a. However, the higher traffic rate leads to more data being processed per batch, causing the elbow to shift to a lower percentile and increasing tail latency across all four engines. For LR4 query, which involves relatively lighter workloads, all four engines display similar patterns. Nevertheless, MiSA achieves up to 4.9× lower 99*th* percentile tail latency compared to the other systems, with a more gradual increase in latency. For Fig. 12c, d, the tail latency of the comparison systems increases more rapidly as traffic grows. In particular, CM1 experiences a sharper increase than LR4 and CM2 at a traffic rate of 2500 records/sec due to the additional computational load caused by the sort operator. In contrast, MiSA exhibits minimal increases in 95*th* and 99*th* percentile tail latencies, maintaining a latency close to its initial levels. For example, in the case of



(a) LR2 (b) LR4 (c) CM1 (d) CM2

Fig. 11 CDF of latencies for MiSA and comparison targets across four queries at a traffic rate of 1500 records/sec on a high-end CPU. An inset figure provides a detailed view of the 95*th* to 99*th* percentile latency. The red dotted-line represents the 99*th* percentile tail latency. Note that figure a uses a different scale on the x-axis. **b**, **c** and **d** share the same y-axis label as figure **a**



(a) LR2 (b) LR4 (c) CM1 (d) CM2

Fig. 12 CDF of latencies for MiSA and comparison targets across four queries at a traffic rate of 2500 records/sec on a high-end CPU. An inset figure provides a detailed view of the 95*th* to 99*th* percentile latency. The red dotted-line represents the 99*th* percentile tail latency As discussed in Sect. 5.2 regarding the LR2 query, figure a includes results only up to the 55*th* batch. Note that figure a uses a different scale on the x-axis. **b**, **c** and **d** share the same y-axis label as figure **a**

CM1, `zStream` and MiSA are 5908 and 439 ms, respectively, a decrease of about 13.4 times at 99*th* percentile tail latency. This stability is achieved by eliminating the commit latency required for state management, allowing the computing time—which previously increased due to the commit task—to remain constant, thereby enabling efficient execution of stateful queries.

It is noteworthy that the tail latency of `Async.Ckp` remains almost identical to the `Baseline` across all queries. For instance, in Fig. 11c, `Async.Ckp` shows slightly better tail latency than the `Baseline` at the 95*th* and 99*th* percentiles, but in Fig. 12c, it is close to the `Baseline`. This indicates that while hiding the stream checkpointing, which is a part of the commit task, may provide temporary improvements, tail latency ultimately increases due to other operations within the commit task. Furthermore, this demonstrates that the primary issue lies in the cumulative overhead caused by sequential dependencies within the commit task. Reducing only the final step, such as stream checkpointing, does not address the underlying problem of delays due to interdependent operations within the commit.

Figure 13 presents the average throughput for the four queries at traffic rates of 1500 records/sec and 2500 records/sec. The average throughput is calculated by dividing the batch size (in bytes) by the time taken to process each batch, averaged over the first 100 batches. Therefore, reducing the latency per batch directly increases throughput.

Overall, MiSA shows significantly higher throughput than its comparison targets across all queries and traffic types. The most dramatic difference is seen in Fig. 13c at a traffic rate of 1500 records/sec, where MiSA achieves 10.4x the throughput of the worst-performing `zStream` (126.58 KB/s vs. 1327.05 KB/s). Even compared to the engine with the highest throughput among the three, `Async.Ckp`, MiSA delivers 4.81× higher throughput (170.99 KB/s vs. 1327.05 KB/s).

It is also worth noting the throughput differences across the queries. In Fig. 13a, at a traffic rate of 1500 records/sec, MiSA achieves a throughput of 337.814 KB/s in LR2. This increases to 1000.939 KB/s (2.96×) in LR4, 1327.05(3.93×) KB/s in CM1 and in 500.92 KB/s (1.48×) CM2. Similarly, the performance improvement of MiSA compared to `Baseline` at the same traffic rate is 1.47× in LR2, rising to 4.87× in LR4, 7.6× in CM1 and 5.1× in CM2. These differences are due to variations in query semantics. LR2 and CM1 involve join and sort operator, respectively, which are more compute-intensive than windowed aggregation in LR4 and CM2. This results in higher batch processing times, which not only reduce throughput but also limit the optimal performance of MiSA, as the

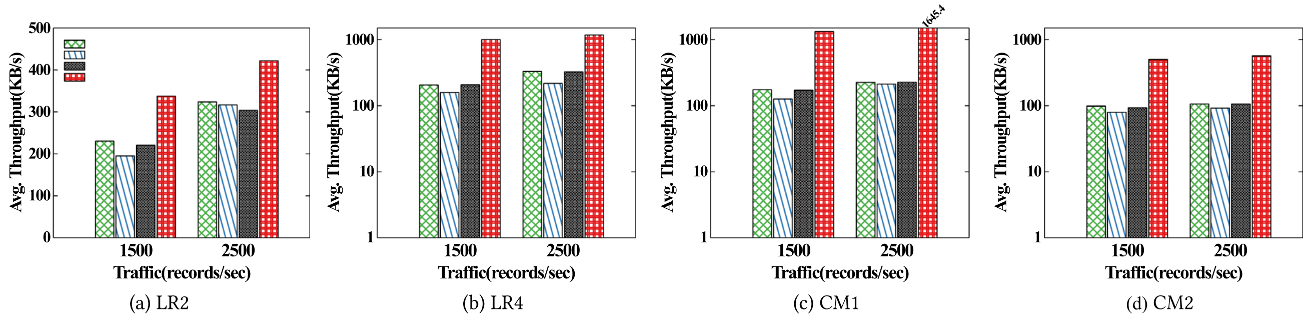


Fig. 13 Average throughput of MISA and comparison targets across four queries under various traffic patterns on a high-end CPU. As discussed in Sect. 5.2, the results in figure a for a traffic rate of 2500 records/sec are limited to the 55*th* batch. Note that the y-axis is presented on a logarithmic scale for LR4, CM1 and CM2. **b**, **c** and **d** share the same y-axis label as figure **a**

longer processing time reduces the opportunity for overlapping query execution and commit operations.

Meanwhile, the high throughput of MISA can be explained by how throughput is calculated. Considering that each record is 65 bytes, the total data ingested is 95.5 KB/sec at a traffic rate of 1500 records/sec. With a trigger interval of 3 s, a single micro-batch buffers approximately 286.5 KB of input data. For a traffic rate of 2500 records/sec, the input data per batch is similarly calculated to be 487.5 KB. Given this, MISA is the only system capable of processing the ingested records within one second per batch, maintaining low latency and high throughput.

5.4 Tradeoff between latency and throughput

To analyze the effect of the trigger value on MISA, we evaluate MISA and all comparison targets across LR2 and LR4 queries by varying the trigger value (or the deadline for `zStream`) from 3 to 5 s. Figure 14 shows the correlation between average latency and throughput for these trigger values. We focus only on the results for LR2 and LR4. In the figure, markers positioned further left and higher up indicate better performance, signifying lower latency and higher throughput.

Overall, Fig. 14 illustrates that MISA consistently exhibits the lowest latency and highest throughput across the two queries, regardless of the trigger value. As the trigger value increases from 3 to 5 s, both average latency and throughput rise across the two queries, which is typical in micro-batch streaming systems. The increase in latency results from the extended buffering time before processing each batch, while the throughput improves due to the larger volume of data processed in each batch.

In LR2, which involves a high-computation workload, MISA consistently achieves higher throughput and significantly lower latency compared to the other targets, regardless of the trigger value. While the three comparison targets exhibit average latencies close to the trigger value,

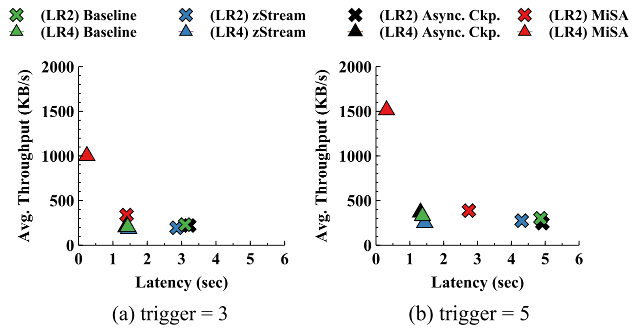


Fig. 14 Correlation between average latency and throughput for LR2 and LR4 on a high-end CPU. For `zStream`, the deadlines are set to 3 and 5 s, respectively

MISA demonstrates significantly lower average latency. However, the increase in throughput is less pronounced than the rise in latency, due to the saturation of computing resources during query execution, while larger batch sizes contribute to higher latency, causing latency to rise more sharply than throughput.

In contrast, LR4 involves a less computationally intensive workload compared to LR2. Like in LR2, MISA achieves the lowest latency and highest throughput in LR4, regardless of the trigger value. However, unlike LR2, where latency increases significantly in relation to throughput, all four systems in LR4 show only a slight rise in latency. While computing resources are saturated in LR2, the underutilization of resources in LR4 allows throughput to increase linearly as the batch size grows.

The rate of throughput increase due to the trigger increment is most pronounced in `Async.Ckp`. For example, when the trigger value shifts from 3 to 5 s, `Async.Ckp`'s throughput increases by 77% compared to a 51% rise for MISA. In MISA, the processing time grows only slightly with the trigger value, resulting in smaller batch accumulation compared to other systems. In contrast, `Async.Ckp` experiences a more substantial increase in

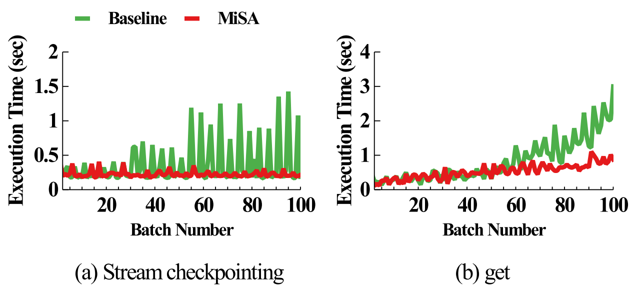


Fig. 15 Execution times for stream checkpointing and get operations per batch. The experiments were conducted using the LR2 query at a traffic rate of 1500 records/sec on a high-end CPU

latency, leading to larger batch volumes and, consequently, a higher rate of throughput growth.

5.5 Effect of state preloading

Figure 15 shows the execution time per batch for stream checkpointing and get operations up to the 100*th* batch. By comparing the performance of these two operations across the `Baseline` and MiSA , we aim to demonstrate that the delay in the commit task is due to the synchronous nature of the commit task and the remote checkpointing location. The location of stream checkpointing in `Baseline` is DFS, while the location of MiSA is the local machine.

Figure 15a illustrates the execution time for the stream checkpointing operation. While the execution time of the `Baseline` approach increases with each compaction cycle, MiSA maintains a constant execution time throughout. Additionally, execution time increases compared to the initial batch due to the large number of files generated during the compaction process compared to a scenario where compaction is not performed. This outcome is expected, as the baseline performs stream checkpointing on a remote DFS, introducing network delays, whereas MiSA performs checkpointing directly on the local machine, thereby avoiding such delays. Furthermore, MiSA's state preloading technique asynchronously copies locally checkpointing files to the DFS, eliminating additional overhead.

Figure 15b depicts the execution time for the get operation. Unlike the trends observed in Fig. 15a, the execution time per batch continues to rise. The rise is due to the growing number of states, which results in longer traversal times to retrieve the necessary state. In contrast to `Baseline`, which experiences a sharp rise in execution time, MiSA maintains stable performance, with its rate of increase significantly lower than that of `Baseline`. This indicates that state preloading effectively reduces execution time for each task while maintaining stability, enabling fast processing and minimizing tail latency.

Table 4 Resource usage (%) of `Baseline` and MiSA on low-end and high-end CPUs

	Low-end CPU		High-end CPU	
	`Baseline`	MiSA	`Baseline`	MiSA
CPU usage (%)	24.57	34.77	7.62	8.43
Memory usage (%)	44.49	12.74	18.37	7.27

5.6 Overhead analysis

Table 4 presents an analysis of CPU and memory usage for both `Baseline` and MiSA on low-end CPU and high-end CPUs, using the LR2 query at a traffic rate of 1500 records/sec.

In a low-end CPU environment, MiSA consumes approximately 30% less memory compared to `Baseline`, while utilizing about 10% more CPU. By executing query and commit tasks concurrently, MiSA leverages CPU resources efficiently, resulting in relatively low overhead.

Conversely, in a high-end CPU environment, the difference in CPU usage between MiSA and `Baseline` is minimal. This is mainly due to the high-performance CPU's ability to effectively mitigate the demands of the CPU-intensive compaction task. Additionally, MiSA consistently exhibits lower memory usage across both low-end and high-end CPUs. Its capability to execute query tasks more quickly than `Baseline` leads to smaller batch sizes, which further contributes to the reduced memory consumption.

In summary, MiSA demonstrates efficient resource utilization, even though its CPU usage rate may not be as high as that of `Baseline`. Furthermore, MiSA shows the ability to deliver strong performance, even on low-end CPUs.

6 Conclusion

This paper addresses the issues of performance degradation when LSM-KVS is utilized as a state store in micro-batch streaming systems. In these systems, the synchronous commit task suspends the streaming engine, causing increasing batch sizes and latency, ultimately degrading system performance. We propose MiSA, which implements asynchronous commit and state preloading in LSM-KVS based micro-batch streaming systems. MiSA hides the time-consuming synchronized commit task within the critical path and performs hierarchical checkpointing to both local and remote nodes. Experimental results show that these mechanisms effectively reduce tail latency while maximizing system throughput with minimal overhead.

Our proposed mechanism has limitations; (1) When compaction overlaps with LSM checkpointing, the resulting wait times reduce the benefits of overlapping commits, offering minimal performance improvement. (2) Performance may degrade as the number of partitions increases due to higher CPU load from additional commit threads. Therefore, carefully configuring the number of partitions is essential when executing MiSA to achieve an optimal balance between performance and resource utilization.

Author Contributions The original draft was written by K.P. (Kyuli Park) and D.J. (Dongjae Lee). Experiments were conducted by Y.J. (Yeonwoo Jeong). S.P. (Sungyong Park) and S.H. (Salim Hariri) provided feedback on the original draft and contributed to manuscript improvement. All authors reviewed and approved the final version of the manuscript.

Funding This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. RS-2024-00453929) (No. RS-2024-00416666) (No. NRF-2021R1A2C2014386).

Data Availability The datasets generated during and/or analysed during the current study are available from the corresponding author on reasonable request.

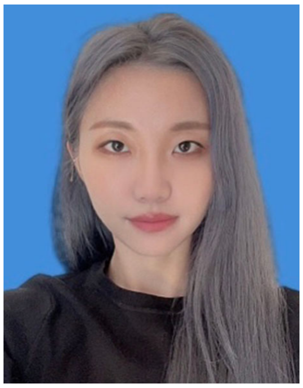
Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.'

References

1. Dai, Y., Yan, J., Tang, X., Zhao, H., Guo, M.: Online credit card fraud detection: a hybrid framework with big data technologies. In: 2016 IEEE Trustcom/BigDataSE/ISPA, pp. 1644–1651 (2016)
2. Chang, G., Zhao, L., Liu, J., Li, P.: An anomaly detection method for stateful stream processing system. In: Proceedings of the 2017 2nd International Conference on Modelling, Simulation and Applied Mathematics (MSAM2017), pp. 196–199 (2017)
3. Noghabi, S.A., Paramasivam, K., Pan, Y., Ramesh, N., Bringhurst, J., Gupta, I., Campbell, R.H.: Samza: stateful scalable stream processing at linkedin. Proc. VLDB Endow. **10**, 1634–1645 (2017)
4. Hazem, H., Awad, A., Yousef, A.H.: A distributed real-time recommender system for big data streams. Ain Shams Eng J **14**(8), 102026 (2023)
5. Gupta, A., Birkner, R., Canini, M., Feamster, N., Mac-Stoker, C., Willinger, W.: Network monitoring as a streaming analytics problem. In: Proceedings of the 15th ACM Workshop on Hot Topics in Networks, pp. 106–112 (2016)
6. Kolokasis, I.G., Papagiannis, A., Pratikakis, P., Bilas, A., Zakkak, F.: Say goodbye to off-heap caches! on-heap caches using memory-mapped i/o. In: Proceedings of the 12th USENIX Conference on Hot Topics in Storage and File Systems, p. 4 (2020)
7. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache Flink^{TM}: Stream and Batch Processing in a Single Engine. The Bulletin of the Technical Committee on Data Engineering. **38**(4) (2015)
8. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., et al.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016)
9. Iqbal, M.H., Soomro, T.R., et al.: Big data analysis: Apache storm perspective. Int. J. Comput. Trends Technol. **19**(1), 9–14 (2015)
10. Thein, K.M.M.: Apache kafka: next generation distributed messaging system. Int. J. Sci. Eng. Technol. Res. **3**(47), 9478–9483 (2014)
11. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.: The log-structured merge-tree (lsm-tree). Acta Informat. **33**, 351–385 (1996)
12. Facebook: Rocksdb: a persistent key-value store for fast storage environment (2012). https://rocksdb.org
13. Google: Leveldb (2017). https://github.com/google/leveldb
14. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (2010)
15. Cheng, D., Zhou, X., Wang, Y., Jiang, C.: Adaptive scheduling parallel jobs with dynamic batching in spark streaming. IEEE Trans. Parallel Distrib. Syst. **29**(12), 2672–2685 (2018)
16. Lee, S., Jeong, Y., Park, K., Jung, G., Park, S.: zstream: towards a low latency micro-batch streaming system. Clust. Comput. **26**(5), 2773–2787 (2023)
17. databricks (2024). https://docs.databricks.com/en/structured-streaming/async-checkpointing.html
18. Armbrust, M., Das, T., Torres, J., Yavuz, B., Zhu, S., Xin, R., Ghodsi, A., Stoica, I., Zaharia, M.: Structured streaming: a declarative api for real-time applications in apache spark. In: Proceedings of the 2018 International Conference on Management of Data, pp. 601–613 (2018)
19. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–10 (2010)
20. Ghemawat, S., Gobioff, H., Leung, S.-T.: The google file system. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles, pp. 20–43. Bolton Landing, NY (2003)
21. Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D.E., Maltzahn, C.: Ceph: a scalable, high-performance distributed file system. In: 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06). USENIX Association, Seattle, WA (2006)
22. Gu, R., Yin, H., Zhong, W., Yuan, C., Huang, Y.: Meces: latency-efficient rescaling via prioritized state migration for stateful distributed stream processing systems. In: 2022 USENIX Annual Technical Conference (USENIX ATC 22), (Carlsbad, CA), pp. 539–556. USENIX Association (2022)
23. Del Monte, B., Zeuch, S., Rabl, T., Markl, V.: Rhino: efficient management of very large distributed state for stream processing engines. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp. 2471–2486 (2020)
24. Lee, G., Maeng, J., Park, J., Seo, J., Cho, H., Yang, Y., Um, T., Lee, J., Lee, J.W., Chun, B.-G.: Flowkv: a semantic-aware store for large-scale state management of stream processing engines. In: Proceedings of the 18th European Conference on Computer Systems, pp. 768–783 (2023)
25. Zhang, S., Wang, Q., Kanemasa, Y., Michaelis, J., Liu, J., Pu, C.: 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, pp. 281–294 (2022)
26. Jayasekara, S., Harwood, A., Karunasekera, S.: A utilization model for optimization of checkpoint intervals in distributed

stream processing systems. Futur. Gener. Comput. Syst. **110**, 68–79 (2020)
27. Geldenhuys, M.K, Thamsen, L., Kao, O.: Chiron: optimizing fault tolerance in qos-aware distributed stream processing jobs. In: 2020 IEEE International Conference on Big Data (Big Data), pp. 434–440 (2020)
28. Balmau, O., Dinu, F., Zwaenepoel, W., Gupta, K., Chandhiramoorthi, R., Didona, D.: SILK: preventing latency spikes in Log-Structured merge Key-Value stores. In: 2019 USENIX Annual Technical Conference (USENIX ATC 19), (Renton, WA), pp. 753–766. USENIX Association (2019)
29. Yu, J., Noh, S.H., Ri Choi, Y., Xue, C.J.: ADOC: automatically harmonizing dataflow between components in Log-Structured Key-Value stores for improved performance. In: 21st USENIX Conference on File and Storage Technologies (FAST 23), (Santa Clara, CA), pp. 65–80. USENIX Association (2023)
30. Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A.S., Ryvkina, E., Stonebraker, M., Tibbetts, R.: Linear road: a stream data management benchmark. In: Proceedings of the 13th International Conference on Very Large Data Bases, vol. 30, pp. 480–491 (2004)
31. Upadhyaya, P., Kwon, Y., Balazinska, M.: A latency and fault-tolerance optimizer for online parallel query plans. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, pp. 241–252 (2011)
32. Sato, K., Mohror, K., Moody, A., Gamblin, T., Supinski, B.R.D., Maruyama, N., Matsuoka, S.: A user-level infiniband-based file system and checkpoint strategy for burst buffers. In: 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 21–30 (2014)
33. Ni, X., Meneses, E., Kalé, L. V.: Hiding checkpoint overhead in hpc applications with a semi-blocking algorithm. In: 2012 IEEE International Conference on Cluster Computing, pp. 364–372 (2012)
34. Di, S., Bouguerra, M.S., Bautista-Gomez, L., Cappello, F.: Optimization of multi-level checkpoint model for large scale HPC applications. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp. 1181–1190 (2014)
35. Reiss, C., Wilkes, J., Hellerstein, J.L.: Google Cluster-Usage Traces: Format + Schema, vol. 1, pp. 1–14. Google Inc., White Paper (2011)
36. Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., et al.: Spark sql: relational data processing in spark. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 1383–1394 (2015)
37. Scala. https://www.scala-lang.org/
38. Scala-future. https://docs.scala-lang.org/overviews/core/futures.html



Kyuli Park received the B.S degree in computer science and engineering from Sogang University, South Korea, in 2022 and the M.S degree in computer science and engineering in Sogang University, in 2024. She is currently pursuing Ph.D. degree in computer science and engineering from Sogang University. She is interested in cloud computing, streaming system and resource management.

Dongjae Lee received the B.S degree in computer science and engineering as part of a double major program from Sogang University, South Korea, in 2024. He is currently pursuing the M.S. degree with the Department of Computer Science and Engineering, Sogang University. His current research focuses on optimizing state management in distributed stream processing systems and its interaction with key-value store.

Yeonwoo Jeong received the B.S. degree in computer software from Kwangwoon University, South Korea, in 2016 and the M.S degree in computer science and engineering in Sogang University, South Korea, in 2020. He is currently pursuing Ph.D. degree in computer science and engineering from Sogang University. His current research focuses on the way of optimizing query processing on distributed stream processing systems.

Salim Hariri is a professor and University of Arizona site director of the NSF-funded Center for Cloud and Autonomic Computing. He founded the IEEE/ACM International Symposium on High Performance Distributed Computing, or HPDC, and is the co-founder of the IEEE/ACM International Conference on Cloud and Autonomic Computing. Professor Hariri serves as editor-in-chief of the scientific journal Cluster Computing, which presents "research and applications in parallel processing, distributed computing systems and computer networks." Additionally, he coauthored three books on autonomic computing, parallel and distributed computing, and edited Active Middleware service, a collection of papers from the second annual AMS workshop published by Kluwer in 2000.

Sungyong Park is a professor in the Department of Computer Science and Engineering at Sogang University, Seoul, Korea. He received his B.S. degree in computer science from Sogang University, and both the M.S. and Ph.D. degrees in computer science from Syracuse University. From 1987 to 1992, he worked for LG Electronics, Korea, as a research engineer. From 1998 to 1999, he was a research scientist at Telcordia Technologies (formerly Bellcore), where he developed network management software for optical switches. His research interests include cloud computing and systems, virtualization technologies, high performance I/O and storage systems, and embedded system software.