Scalable Transactional Stream Processing on Multicore Processors

Jianjun Zhao, Yancan Mao, Zhonghao Yang, Haikun Liu, Shuhao Zhang

***Abstract*—Transactional stream processing engines (TSPEs) are central to modern stream applications handling shared mutable states. However, their full potential, particularly in adaptive scheduling, remains largely unexplored. We present *MorphStream*, a TSPE designed to optimize parallelism and performance for transactional stream processing on multicores. Through a unique three-stage execution paradigm (i.e., *planning*, *scheduling*, and *execution*), *MorphStream* enables adaptive scheduling under varying workload characteristics. Building on this foundation, *MorphStream* is further enhanced with support for non-deterministic state access, employing a stateful task precedence graph to handle undefined read/write sets at runtime while guaranteeing transaction semantics. Additionally, *MorphStream* incorporates a generalized framework for managing window-based operations, enabling efficient tracking and maintenance of overlapping windows using multi-versioned state management. These extensions enhance the system's ability to process dynamic and irregular workloads. Experimental results demonstrate up to 3.4 times higher throughput and 69.1% lower latency compared to state-of-the-art TSPEs, validating its scalability and adaptability in real-world streaming scenarios.**

***Index Terms*—Stream processing, transaction, multicore.**

I. Introduction

THE advent of real-time stream processing, driven by the growth of data-intensive applications and the proliferation of the *internet of things* (IoT), promotes the development of *stream processing engines* (SPEs) such as Storm [1], Flink [2], and Spark-Streaming [3]. However, since many emerging stream applications [4], [5], [6], [7] involve shared mutable states that are read and modified concurrently by multiple execution entities, mainstream SPEs face significant challenges related to correctness [8] and efficiency [9], [10]. To address these concerns, academia and industry have recently turned their attention to *transactional stream processing engines* (TSPEs) [10], [11], [8], [4], which offer built-in support for shared mutable states.

Unlike conventional SPEs, TSPEs employ transactional semantics to manage concurrent accesses to shared mutable states during continuous data stream processing. In TSPEs, a set of state access operations triggered by a single input event is modeled as a state transaction. Subsequently, the concurrent process of state transactions must be scheduled to ensure the order of the streaming events and ACID properties (formally discussed in Section II). Despite substantial efforts in the field, current state-of-the-art TSPEs [8], [10] rely mainly on some non-adaptive task scheduling strategies. S-Store [8] uses *state partitioning* to reduce context switching overhead and can handle transaction abort as early as it occurs. However, as parallelism increases, the intensification of access conflicts among state partitions severely limits the system's performance. TStream [10] outperforms S-Store [8] on multicore architectures with a *transaction restructuring* paradigm, which decomposes state transactions into atomic operations. Although this approach reduces state access conflicts, it affects the efficient handling of transaction abort and complex data dependencies in the workload.

Thus, none of the existing TSPEs can maximize performance under different and dynamically changing workload characteristics, leaving a significant design space for scaling TSPEs on multicore processors largely unexplored. Moreover, current TSPEs lack comprehensive support for non-deterministic operations, such as those involving randomization or external data sources, which are crucial in many real-world applications. Additionally, sophisticated window-based computations, including sliding and tumbling windows with intricate aggregation or join operations, are either inefficiently handled or entirely unsupported, further limiting their applicability in diverse streaming scenarios.

In this paper, we present *MorphStream*, a TSPE designed to optimize parallelism and performance for transactional stream processing on multicores. *MorphStream* utilizes a *task precedence graph* (TPG) that identifies the fine-grained dependencies among a batch of state transactions. Based on the TPG, *MorphStream* can adaptively morph multiple scheduling strategies under varying workload characteristics, including non-deterministic and window-based operations. To ensure both efficiency and correctness, *MorphStream* employs a three-stage execution paradigm:

- **Planning:** *MorphStream* leverages a two-phase parallel TPG construction mechanism to efficiently track data dependencies among state transactions, including non-deterministic and window-based operations.
- **Scheduling**: Based on TPG, *MorphStream* decomposes scheduling strategies into three dimensions and strives to make the right decision along each dimension by analyzing trade-offs under varying workload characteristics, ensuring adaptive and efficient scheduling.
- **Execution**: *MorphStream* ensures correct execution of the state transaction based on scheduling decisions. It uses the stateful TPG to manage transaction lifecycles and

Jianjun Zhao and Yancan Mao make equal contributions.

Jianjun Zhao, Haikun Liu and Shuhao Zhang are affiliated with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China. Yancan Mao is with the National University of Singapore, Singapore. Zhonghao Yang is with Nanyang Technological University, Singapore.

employs multi-versioned state management to maintain shared mutable state consistency and correctness.

A preliminary version of this work was recently published in SIGMOD 2023 [9], where we introduced an adaptive scheduling strategy for concurrent state transactions in *MorphStream*. However, many critical execution and implementation details were omitted. This paper expands on those details and showcases the full capabilities of *MorphStream*, emphasizing its scalability in processing state transactions through a three-stage execution paradigm that extends beyond adaptive scheduling. Building on the foundational work, we enhance *MorphStream* with advanced features for non-deterministic state access and window-based operations, which were not covered in the conference version [9]. Specifically, *MorphStream* employs a proactive strategy to anticipate potential state access, improving dependency tracking and guaranteeing transaction semantics even when read/write sets remain undefined until runtime. Additionally, *MorphStream* introduces a generalized framework for managing window-based operations, enabling efficient tracking and maintenance of overlapping windows through multi-versioned state management.

We evaluated *MorphStream* on a dual-socket Intel Xeon Gold 6248R server with 384 GB of DRAM, focusing on a single socket to isolate NUMA effects. Our experiments show that *MorphStream* achieves up to 3.4× higher throughput and 69.1% lower latency when handling real-world use cases with complex, dynamically changing workload dependencies. We further showcase *MorphStream*'s versatility through two innovative applications: online social event detection [6] and stock exchange analysis [12]. *MorphStream* achieved a throughput of up to 1.3k tweets per second and processed up to 70k stock events per second, highlighting its performance advantages and broad applicability.

Overall, this work makes the following contributions:

- We introduce *MorphStream*, a novel TSPE specifically designed for multicore CPU architectures. *MorphStream* features a three-stage execution paradigm—planning, scheduling, and execution—that enables adaptive scheduling, optimizes parallelism, and ensures efficient, low-latency transactional stream processing.
- *MorphStream* extends the capabilities of TSPEs by providing robust support for non-deterministic state access and efficient window-based operations. These enhancements leverage the three-stage execution paradigm to address key challenges in dependency tracking and overlapping window management.
- We extensively evaluate *MorphStream* 's performance, showing substantial improvements in throughput and latency over state-of-the-art systems. We further validate *MorphStream* 's utility in real-world applications, such as online social event detection [13], [14] and real-time stock exchange analysis [15].
- To foster transparency and inspire future research, we release *MorphStream* 's codebase, along with associated datasets and scripts, at https://github.com/intellistream/MorphStream. This contribution lays a foundation for advancing transactional stream processing in both academia and industry.

II. Background and Motivation

In this section, we present preliminaries of transactional stream processing and motivations for optimizing TSPEs.

A. Transactional Stream Processing

In modern *stream processing engines* (SPEs), operators are structured as a *directed acyclic graph* (DAG), where each vertex represents an operator and each edge represents an event flowing downstream from one operator to another. To handle high-throughput streams, operators are typically parallelized across multiple executors (e.g., Java threads), each processing a subset of the input data via key-based partitioning [1], [3], [16]. However, the strict partitioning of states introduces challenges when processing workloads that require transactional cross-partition state access. Since the partitioning strategy is determined before execution, operations that need to update state spanning multiple partitions often require expensive workarounds, such as routing data between partitions or maintaining state in external storage [8], [10].

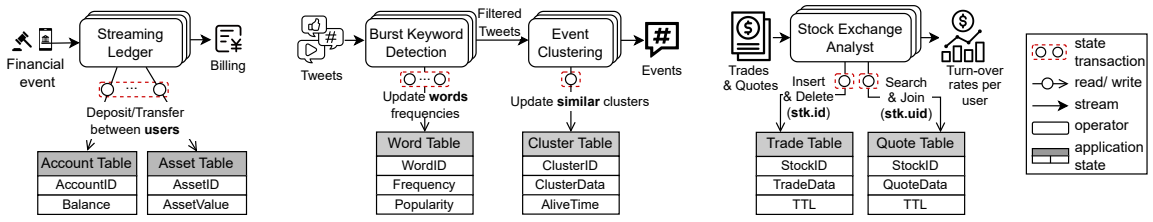
These inefficiencies drive growing interest in *transactional stream processing engines* (TSPEs) from both academia [5], [8], [7], [10], [11], [17], [9] and industry [4], [6]. In contrast to conventional SPEs, TSPEs maintain a shared mutable state, which can be referenced and updated by multiple threads spawned from the same stream application. To ensure transactional properties, the concurrent accesses (i.e., read and write) to the shared mutable states must satisfy the following predefined constraints.

We define a *state access operation* as either a read ($\underline{O_i = Read_{ts}(k)}$) or a write ($\underline{O_i = Write_{ts}(k, v)}$) operation on a shared state. Here, ts represents the timestamp of the triggering event, k is the key for the state to be accessed, and v is the value to be written. The value v may be computed as $v = f(k_1, k_2, ..., k_m)$, where f is a user-defined function that only reads state values [10], [8].

A *state transaction* is a set of state access operations triggered by one input event, expressed as $txn_{ts} =< O_1, ... O_n >$. A schedule ($S$) of such state transactions ($txn_{t1}, ..., txn_{tn}$) is deemed correct if it is *conflict equivalent* to an ordered sequence $txn_{t1} \prec txn_{t2} \prec ... \prec txn_{tn}$, where $\prec$ indicates that one transaction precedes another in the order.

Scaling *transactional stream processing* (TSP) is crucial for handling high input stream rates. Therefore, TSPEs aim to maximize concurrency while ensuring a correct schedule. This challenge arises from the complex inter- and intra-dependencies among state transactions. Below, we outline three key workload dependencies in TSP applications:

- **Temporal Dependency (TD):** O_i temporally depends on O_j if they are from different state transactions, access the same state, and O_i has a larger timestamp. Tracking TD enforces that state accesses follow the event sequence.
- **Parametric Dependency (PD):** $O_i = Write(k_i, v)$, where $v = f(k_1, k_2, ..., k_m)$, parametrically depends on $O_j = Write(k_j, v')$ if $k_j \neq k_i$, $k_j \in k_1, k_2, ..., k_m$, and



(a) Streaming Ledger (SL) (b) Online Social Event Detection (OSED) (c) Stock Exchange Analysis (SEA)

Fig. 1: Representative examples. Note that, due to the complexity of OSED and SEA, this figure focuses solely on shared mutable states to illustrate the necessity of TSPE, with the detailed application workflow elaborated in Section VIII-C.

O_i has a larger timestamp. Tracking PD resolves the potential conflicts among write operations due to user-defined functions.

- **Logical Dependency (LD):** O_i and O_j logically depend on each other if they belong to the same state transaction. Tracking LD ensures that aborting one operation leads to aborting all operations within the same transaction.

B. Motivation Examples

TSPEs are crucial for modern applications that demand high-performance, stateful, and transactional guarantees under dynamic and complex workloads. For example, streaming ledgers demand transactional integrity to process concurrent money transfers and deposits correctly. Online social event detection relies on transactional updates to manage unpredictable state access patterns for clustering and burst keyword detection. Similarly, stock exchange analysis requires transactionally consistent state management for overlapping window-based aggregations of trades and quotes. These diverse scenarios highlight the limitations of existing TSPEs, which struggle to effectively exploit parallelism, adapt to varying workloads, and support non-deterministic state access and window-based operations. Figure 1 provides a visual summary of these examples, which motivate the need for advanced TSPE capabilities.

Streaming Ledger. Figure 1a illustrates a running example of a simplified use case that benefits from TSPEs, namely *Streaming Ledger* (SL), which was suggested by a recently announced commercialized version of Apache Flink [4]. Unlike batch-based ledger systems, it processes a stream of requests involving **wiring money and assets among users** and outputs the processing results as an output stream to users. Two types of state transactions accessing two tables of shared mutable states are generated during the processing of each input request: (1) a *Transfer* transaction processes a request that transfers balances between user accounts and assets, and (2) a *Deposit* transaction processes requests that top-up user accounts or assets. The processing of all concurrent state transactions needs to ensure transactional semantics. For instance, a state transaction may be aborted due to the violation of a consistency property, such as that the account balance can not become negative.

Online Social Event Detection. Modern stream processing frequently manages unpredictable data streams from dynamic sources, such as IoT devices, social media feeds, and online interactions [18], [19], [20], where state access keys cannot always be predetermined. A notable example is *online social event detection* (OSED) [13], [14], which identifies trending events in real-time tweet streams. OSED involves two stages: burst keyword detection and event clustering, as shown in Figure 1b. Each stage comprises multiple streaming operators, where each operator performs a specific state access task and is distributed across multiple parallel threads for concurrent execution. Each thread executes read or write operations on key-value state storage entities, such as Word and Cluster tables. For example, each tweet is decomposed into **several keywords**, and OSED **transactionally updates** the frequency of these words to ensure accurate burst keyword detection. Moreover, Event Clustering aims to find the most similar cluster through similarity calculations and subsequently update the relevant cluster. As a result, the read/write sets (e.g., clusterID) **cannot be determined in advance** until the similarity calculations are completed.

Stock Exchange Analysis. Window-based operations play a critical role in temporal data aggregation and processing within TSPEs, introducing substantial complexity to their implementation. These operations, essential for applications like *stock exchange analysis* (SEA) [12], involve the continuous computation of statistics, such as maximum, minimum, average and count, across streaming data, requiring precise state management within moving windows. As shown in Figure 1c, SEA calculates portfolio turnover rates [15] for each user by processing streams of quotes and trades within specified temporal windows. This process necessitates maintaining a trade/quote state table for each stock while simultaneously performing **global searches and joins** on stock groups by users over windows. Accurate and **transactionally consistent read/write operations** on quotes and trades must be maintained for each window, posing significant challenges in managing overlapping window aggregations and dynamic state updates effectively. The complexity of these requirements—particularly with sliding windows that shift continuously over time—underscores the importance of robust TSPE capabilities to ensure data accuracy and consistency under diverse operational conditions.

C. Programming Model

In *MorphStream*, TSP applications are depicted as a DAG, where each vertex represents an operator within a dataflow framework. *MorphStream* adopts a programming model similar to TStream [10] and S-Store [8], enabling fine-grained performance optimizations by allowing each

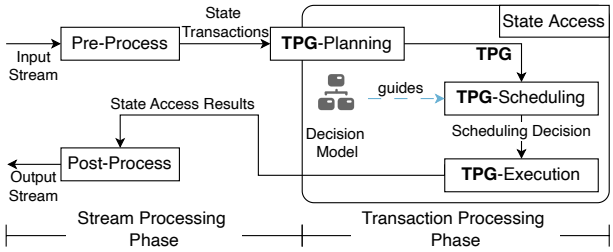


Fig. 2: The execution workflow of *MorphStream* atop the programming model.

operator to generate its own transaction for each incoming event. Users can customize the user-defined APIs to meet their application-specific needs, while the system-integrated APIs operate like standard library functions. As shown in Figure 2, the user-implemented APIs require users to follow a three-step procedure to implement the operations of an operator: First, *preprocessing* the input events to identify the read/write sets of state transactions. For non-deterministic state access where exact values and states accessed may not be immediately known, *MorphStream* uses predefined logic to anticipate potential read/write sets. Second, performing *state access*, where all state-related operations are expressed through system-provided APIs, including non-deterministic and window-based state access. Finally, *post-processing* is conducted to further process the input events based on the access results and generate corresponding outputs. The catalog of system-integrated APIs supports atomic operations like READ, WRITE, and can extend to more complex non-deterministic and window-based operations. Once an application is fully sculpted, it is dispatched as a job to *MorphStream*, priming it for execution.

In the last decade, several TSPE variants [10], [11], [8], [21] with different programming models have emerged, aiming to balance developer simplicity and efficiency in combining real-time stream processing with transactional guarantees [22]. Early SPEs like STREAM [23], [24] used a relational query model to integrate streaming and transactional operations, simplifying transaction management by grouping all operations in a query into a single transaction. However, this lacked flexibility, especially when different operators within a query required separate transactional boundaries or isolation levels. More recently, FlowDB [11] and TSpoon [21] introduced user-defined transactions, allowing users to specify custom transactional boundaries and consistency constraints based on their specific application needs. While this provided more flexibility, it also increased system complexity, requiring developers to balance performance and correctness and making system optimization more challenging. In contrast to these approaches, *MorphStream* adopts a programming model that not only provides flexibility but also enables more fine-grained performance optimizations. As discussed earlier, this model allows each operator to independently generate its transactions in response to incoming events.

III. Design of MorphStream

In this section, we outline the design principles underpinning *MorphStream*, and describe its three-stage execution paradigm and user-friendly APIs.

A. Design Principles

MorphStream adopts a novel adaptive scheduling strategy by formulating the state transaction scheduling problem as a graph scheduling problem. Specifically, *MorphStream* constructs a *task precedence graph* (TPG), where each vertex uniquely represents a state access operation, and edges capture the data dependencies among these operations, as outlined in Section II. Ensuring a correct schedule involves processing all operations while adhering to these data dependencies. Building on this foundation, *MorphStream* further addresses complex challenges, including non-deterministic and window-based operations, which remain significant hurdles for existing TSPEs. Thus, the key principles guiding *MorphStream* are as follows:

- *Efficient dependency management*: *MorphStream* should accurately capture dependencies among state transactions, especially in scenarios involving non-deterministic and window-based operations.
- *Adaptability to dynamic workloads*: Unlike non-adaptive scheduling strategies, *MorphStream* should dynamically adjust its scheduling strategy to accommodate varying workload characteristics.
- *Robust state management: MorphStream* must maintain the consistency and correctness of shared mutable states, especially for ranged read/write in window operations and correctly state rollback for non-deterministic operations.

B. Three-stage Execution Paradigm

Based on the programming model and the design principles outlined earlier, *MorphStream* employs a novel three-stage execution paradigm centered on TPG as shown in Figure 2. The execution workflow is recursively conducted for every batch of input events. *MorphStream* separates the execution workflow into two phases: stream processing and transaction processing. It periodically switches between the two phases, separated by punctuations [25]. A batch of input events is preprocessed during the stream processing phase and transformed into state transactions. These state transactions are issued but are not immediately processed. During the transaction processing phase, the batched state access is then executed through the following three stages:

1) ❶ *Planning: MorphStream* identifies the fine-grained temporal, logical, and parametric dependencies among a batch of state transactions to construct the corresponding TPG in parallel. Note that, TPG planning spans both the stream and transaction processing phases, forming its two-phase TPG construction process. During this process, specific dependency tracking rules are applied to handle non-deterministic and window operations (Section IV).
2) ❷ *Scheduling:* In this stage, *MorphStream* utilizes the constructed TPG to determine the execution order of operations. *MorphStream* decomposes the scheduling based on the TPG into three dimensions of scheduling decisions. It leverages a heuristics decision model to make a suitable scheduling decision based on the current workload characteristics at runtime (Section V).
3) ❸ *Execution:* Threads execute operations concurrently based on the scheduling decisions while ensuring state

TABLE I: System-provided APIs (parameters for `table`, `timestamp`, and `EventBlotter` are omitted)

APIs	Description
READ (Key d, EventBlotter eb)	Initiates a read request for the key d and stores the results in eb for further post-process.
WRITE (Key d, Fun f^* (Keys $s...n$))	Initiates a write request so that $state(d)$ is updated with the results after applying f^* on $state(s...n)$, representing a data dependency.
READ (Window_Fun win_f^* (Key d, Size t), EventBlotter eb)	Issues a window read request that applies win_f^* on a key d with size t and stores the results in eb for further processing (i.e., post-process).
WRITE (Key d, Window_Fun win_f^* (Keys $s...n$, Size t))	Initiates a window write request so that $state(d)$ is updated with the results after applying win_f^* on $state(s...n)$ with size t.
READ (Fun f^*, EventBlotter eb)	Issues a non-deterministic read request on a key determined by a user-defined function f^* and stores result in eb for further processing (i.e., post-process).
WRITE (Fun $f1^*$, Fun $f2^*$)	Initiates a non-deterministic write request where the key to be updated is determined by a user-defined function $f1^*$ and the value to be written back is determined by another user-defined function $f2^*$.

TABLE II: User-implemented APIs

APIs	Description
PRE_PROCESS (Event e)	Implements a pre-processing function (e.g., filtering). Returns an EventBlotter containing parameter values (e.g., read/write sets) extracted from e.
STATE_ACCESS (EventBlotter eb)	Facilitates state transactions by constructing system-provided APIs, such as READ and WRITE.
POST_PROCESS (Event e, EventBlotter eb)	Implements a post-processing function that depends on the results of state access stored in the eb.

access correctness. *MorphStream* employs a finite state machine for each operation to accurately capture its state access behavior during execution and aborting. It relies on the multi-versioning state table management to maintain the consistency of table entries. *MorphStream* supports window-based state access by querying a range of targeting state copies from the multi-versioning state table. Furthermore, it records state accesses for non-deterministic state transactions after execution to ensure accurate rollback if necessary (Section VI).

Finally, after executing transactions, *MorphStream* performs post-processing, generating output streams based on the results of state access operations.

C. User-friendly APIs atop the Programming Model

To enhance the programming model discussed in Section II, *MorphStream* provides a suite of user-friendly APIs that allow both system-provided and user-implemented functionalities, summarized in Table I and II: System-provided APIs support basic operations like READ and WRITE, and extend to more complex non-deterministic and window-based operations. These APIs help define window aggregation functions and specify user-defined functions for keys and values, aligning with the system's capability to handle dynamic transaction adjustments. In particular, for window-based operations, users can define a window aggregation function in READ/WRITE operations, specifying states to access, and window size. Note that the window trigger is configured during execution aligning with the operation's timestamp. *MorphStream* supports non-deterministic operations by allowing users to specify user-defined functions (f^*) for both keys to access and values to write back. This indicates that while the number of operations a transaction decomposes is deterministic according to the processing logic, the accessed keys and aggregated results of operations can be non-deterministic.

Algorithm 1: Code template of an operator

```
1  Map cache;// thread-local storage
2  TPG G;// thread-shared tpg
3  foreach event e in input stream do
4      if e is not punctuation then
5          EventBlotter eb ← PRE_PROCESS(e);
6          txn_ts ← STATE_ACCESS(eb);// issue one state
               transaction
7          TPG_Planning_Phase_1(txn_ts);
8          cache.add(< e, eb >);// stores events whose
               state access is postponed
9      else
10         TXN_Start();// switch to transaction
               processing
11         foreach < e, eb > ∈ cache do
12             POST_PROCESS(< e, eb >);
13 Function TXN_Start():
14     TPG_Planning_Phase_2(G);
15     M ← Instantiated with G;// A decision model
16     while !finish scheduling of G do
17         TPG_Execution() ← TPG_Scheduling(G, M);
```

Algorithm 2: STATE_ACCESS of Streaming Ledger

```
   Input: EventBlotter eb
1  Function Sender_Fun(sender, value):
2      if READ(sender, eb) > value then
3          return READ(sender, eb) - value;
           // decrement money of sender by value.
4  Function Recver_Fun(recver, sender, value):
5      if READ(sender, eb) > value then
6          return READ(recver, eb) + value;
           // increment money of recver by value.
7  begin
8      WRITE(eb.sender, Sender_Fun(eb.sender, eb.v));
9      WRITE(eb.recver, Recver_Fun(eb.recver, eb.sender, eb.v));
```

As discussed in Section II, user-implemented APIs require users to customize the three-step data processing logic to suit their application-specific needs, ensuring seamless integration with *MorphStream*'s operational framework. An illustrative template for implementing a user-defined operator is provided in Algorithm 1. Note that, state transactions are defined through the STATE_ACCESS API, which leverages system-provided APIs, such as READ and WRITE. To provide further clarity, Algorithm 2 demonstrates how the STATE_ACCESS API is implemented, using Streaming Ledger as a representative example. All operations invoked within a single STATE_ACCESS call (e.g., the two operations illustrated in Algorithm 2) are collectively treated as a single state transaction to ensure atomicity and consistency.

IV. Planning

In this section, we detail how *MorphStream* tracks dependencies among state transactions during TPG planning.

A. TPG Construction and Dependency Tracking

Figure 3 illustrates the TPG construction workflow in *MorphStream*, which is divided into two primary phases:

Stream Processing Phase: In this phase, incoming data is preprocessed to prepare state transactions for execution. As state transactions arrive, they are broken down into atomic

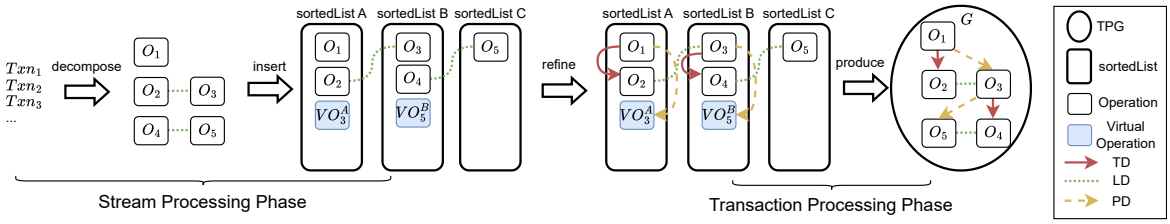


Fig. 3: TPG construction workflow, which is divided into stream processing and transaction processing phase.

state access operations, which serve as the vertices of the TPG. *Logical dependencies* (LDs) are identified among operations from the same transaction according to their statement orders. To manage *temporal dependencies* (TDs) resulting from out-of-order processing due to parallel execution, all operations are inserted into key-partitioned sorted lists based on their target states and timestamps. For each write operation with multiple states, virtual operations are maintained and inserted into the sorted lists in preparation for identifying *parametric dependencies* (PDs) during the next phase.

Transaction Processing Phase: After stream processing, *MorphStream* periodically switches to transaction processing, which is controlled by punctuation. The punctuation is a timestamp marker, ensuring that subsequent input events do not have smaller timestamps. We can now identify TDs and PDs efficiently for this batch of state transactions with the help of the constructed sorted lists and virtual operations during the stream processing phase. First, TDs can be identified straightforwardly by iterating through operations inserted into each sorted list. Note that, virtual operations are not involved in identifying TDs. Second, PDs can be identified according to the inserted virtual operations.

Running Example. Figure 3 shows an example involving three transactions that arrive consecutively. In the stream processing phase, upon arrival, txn_1 is decomposed into O_1, txn_2 into O_2 and O_3, and txn_3 into O_4 and O_5. LDs among O_2, O_3 and O_4, O_5 are identified as they are from the same transaction. A preliminary TPG is constructed by inserting these operations as vertices and the LDs as edges. The operations are then inserted into two sorted lists (one for each state, A, B and C). For operation O_3 and O_5, which have write functions dependent on states A and B, virtual operations (VO_3^A and VO_5^B) are inserted into the sorted lists of states A and B, respectively. During the transaction processing phase, TDs can be identified among O_1, O_2 and O_3, O_4 by iterating through the operations in each sorted list. PDs are identified between O_1 and VO_3^A, and between O_3 and VO_5^B according to the previously inserted virtual operation. After the identification of TDs and PDs, they are inserted as edges to refine the preliminary TPG to become the final TPG of the current batch of state transactions.

B. Tracking dependencies for Non-deterministic Operations

Tracking dependencies of non-deterministic operations is challenging, as the precise read/write set is unknown before execution. In cases where a non-deterministic operation might interact with any state, a pessimistic approach assumes it depends on all preceding operations, improving system robustness but reducing parallel execution efficiency.

Dependency Definitions: Dependencies for non-deterministic operations are defined as follows:

- *Temporal and Parametric Dependencies:* A non-deterministic operation non_O_i is considered dependent on another operation O_j if non_O_i has a later timestamp than O_j. Conversely, any operation O_k that follows non_O_i in time may also depend on non_O_i.
- *Logical Dependencies:* If non_O_i and non_O_j originate from the same state transaction, they are logically interdependent.

To effectively track these dependencies, *MorphStream* integrates virtual operations during the construction of the TPG. When transactions are broken down into atomic operations, LDs are identified. Virtual operations representing non-deterministic accesses are then inserted into the TPG to manage potential PDs and TDs, allowing for precise scheduling and execution of these complex transactions.

Running Example: Figure 4a illustrates an example of tracking dependencies for non-deterministic operations. Consider five atomic state access operations $O_1 \sim O_5$ across three sorted lists: A, B, and C. A new non-deterministic operation O_6 is introduced, which writes a value v determined by a user-defined function UDF. To track dependencies, virtual operations VO_6^A, VO_6^B, and VO_6^C are integrated into the respective sorted lists. The system then assesses each list to determine O_6's dependencies, ensuring accurate and efficient processing of this non-deterministic transaction.

However, when the number of non-deterministic operations becomes large, their corresponding virtual operations dominate each sorted list, leading to increased overhead for dependency tracking. This overhead arises because the virtual operations for non-deterministic operations must be inserted into every sorted list, requiring repeated iterations over the virtual operations already present in each list. To mitigate this, we maintain a single globally sorted list of virtual operations. This list is then merged with each sorted list to efficiently identify dependencies, minimizing repeated insertions and iterations of virtual operations across all sorted lists.

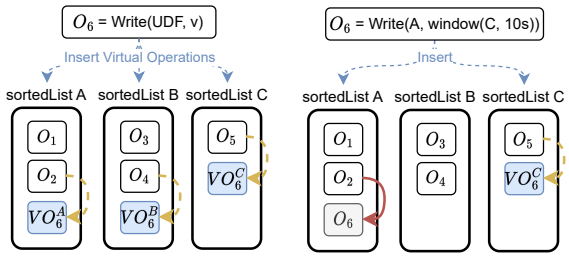
C. Tracking dependencies for Window-based Operations

Tracking dependencies among window-based operations poses significant challenges due to data dependencies that often span multiple, potentially overlapping windows of varying sizes. To address this, *MorphStream* introduces a generalized structure for window-based operations and extends dependencies definitions to identify dependencies based on overlapping window ranges.

In alignment with previous work [26], a window-based operation in *MorphStream* is modeled as a read or write state

TABLE III: Scheduling decisions at three dimensions

Dimension	Decision	Pros	Cons
Exploration Strategy	*s-explore*	Threads can run in parallel with minimum coordination	BFS: Sensitive to workload imbalance / DFS: High memory access overhead
	ns-explore	More parallelism opportunities	Higher message-passing overhead
Scheduling Granularity	*f-schedule*	Better system scalability	High context switching overhead
	c-schedule	Lower context switching overhead	Less scalable and more sensitive to load imbalance
Abort Handling	*e-abort*	Less wasted computing efforts	High context switching overhead
	l-abort	Less context switching overhead	More wasted computing efforts



(a) Non-deterministic (b) Window-based

Fig. 4: Dependencies tracking for two special cases.

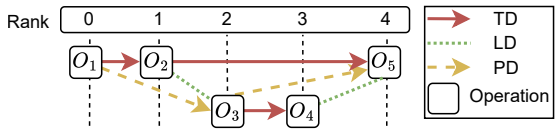


Fig. 5: A stratified auxiliary structure of TPG.

access operation. There are varying types of windows [27], distinguished by their trigger policies (i.e., policies to initiate aggregation) and window maintenance policies (i.e., policies to insert or remove tuples from a window). Regardless of the variety, all window-based operations can be generally structured around two primary attributes: 1) a window size to identify tuples belonging to the window; and 2) a trigger time to indicate when to initiate aggregation. Using this framework, *MorphStream* identifies the window range of a window-based operation by its window size and associated trigger time. This further helps track dependencies among window-based operations. We formally denote a window-based operation as $win\_txn_{ts} =< O_1, O_2, ...O_n >$, where each operation $O_i = Read_{ts}(agg\_v)$ *or* $Write_{ts}(k, agg\_v)$. Here, the $agg\_v = window(k_1, k_2..., k_m, l)$ is the aggregated value calculated by applying a user-defined, read-only aggregation function to a subset of states $< k_1, k_2, ..., k_m >$ within the window size l, which triggers at timestamp ts. Note that the window size can be either count-based or time-based, determined by window maintenance policies of different window types.

Dependency Definitions: Through our generalized structure, dependencies among window-based operations are determined by carefully examining their trigger times and window sizes. This analysis allows for the identification of conflicting read or write, facilitating the construction of a TPG and efficient scheduling of window-based operations. We define a window-based operation O_i temporally or parametrically depends on another operation O_j as follows:

- O_i temporally depends on O_j if: 1) they are not from the same state transaction but access the same state; and 2) the window trigger of O_i has a larger timestamp than that of O_j.
- $O_i = Write_{ts}(k_i, agg\_v)$, where $agg\_v = window(k_1, k_2..., k_m, l)$, parametrically depends on $O_j = Write(k_j, v)$, where v can be either a normal user-defined function or window aggregation function: 1) its states to be accessed overlap with states in another operation, i.e., $k_j \in k_1, k_2, ..., k_m$; and 2) the timestamp of O_j belongs to the window range of O_i. This ensures a window-based operation captures all tuples belonging to its window range.

In addition to supporting tumbling and sliding windows, *MorphStream* also provides native support for session windows. Session windows, defined by their dynamic boundaries based on activity gaps, present unique challenges. These include the need to dynamically adjust window ranges as new events arrive and handle overlapping states caused by session merging. *MorphStream* addresses these challenges through its multi-version state management (Section VII-C) and advanced dependency tracking mechanisms. The system's ability to maintain state versions in temporal order allows efficient merging and splitting of sessions without incurring significant computational overhead. For dependency tracking of session windows, *MorphStream* generates a window-based operation only when a window is triggered for execution, ensuring that the window size is deterministic at the time of execution. This design allows *MorphStream*'s dependency tracking to naturally support session windows, handling their dynamic nature with precision and efficiency.

Running Example. A running example of dependency tracking among window-based operations is provided in Figure 4b. The objective, similar to that in Section IV-B, is to identify dependencies for a new window-based operation O_6 among five operations in three $sortedLists$ A, B, C. Specifically, O_6 is a write operation, tasked to aggregate state access of C within the past 10 seconds and write the results to the target state A. Following the dependency tracking mechanism, we insert operation O_6 into the $sortedList$ A and a virtual operation VO_6^C into the $sortedList$ C. Subsequently, based on our dependency tracking definitions, we establish a PD between O_5 and O_6, and a TD between O_2 and O_6.

V. Scheduling

MorphStream decomposes the scheduling based on the TPG into three dimensions, namely exploration strategies, scheduling unit granularities, and abort handling mechanisms. Each of these dimensions assists *MorphStream* in adapting to different workload patterns and system states. Table III summarizes potential decisions within each dimension. To optimize these dynamic scheduling decisions, we further introduce a heuristic decision model. This model evaluates the current workload characteristics at runtime, enabling appropriate scheduling decisions (Section V-D).

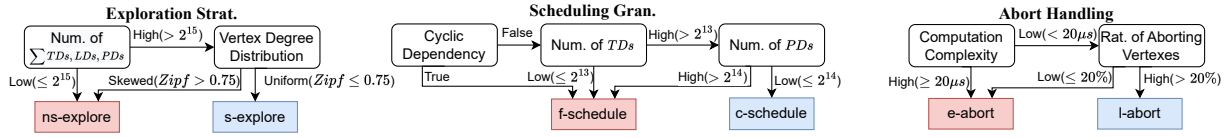


Fig. 6: The lightweight decision model. The concrete threshold numbers in brackets are based on our experiments.

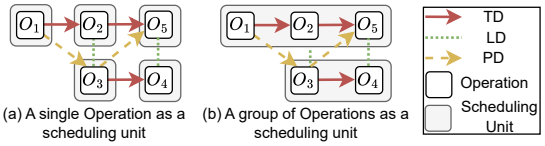


Fig. 7: Different scheduling unit granularities.

TABLE IV: Workload characteristics to TPG Properties

Type	TPG Prop.	Workload Char.
Vertex	Computation Complexity	C
	Vertex Degree Distribution	$\underset{\sim}{\propto} \theta$
	Ratio of Aborting Vertexes	$\underset{\sim}{\propto} \alpha$
Edge	Number of LDs	$\underset{\sim}{\propto} T * l$
	Number of TDs	$\underset{\sim}{\propto} T * l$
	Number of PDs	$\underset{\sim}{\propto} T * l * r$
	Cyclic Dependency	Correlated to θ, T, l, r

A. Exploration Strategies

This dimension defines how the TPG is traversed for scheduling operations, using either structured or unstructured exploration strategies.

In the structured approach (*s-explore*), we first generate an auxiliary structure as shown in Figure 5. Specifically, vertices are partitioned into subsets based on connected directed paths, as described by Nikolov et al. [28], with subsets receiving ranks for stratum placement. This structured exploration bifurcates into: *A) BFS-like Exploration*, where threads concurrently explore operations within the same stratum, moving to the next only after all current operations are processed, dependent on barrier-based synchronization, jeopardized by uneven thread workloads; and *B) DFS-like Exploration*, threads are assigned an equal number of operations in each stratum and can move to the next stratum once their dependencies are resolved. This approach reduces synchronization overhead by allowing threads to progress independently but may increase memory access overhead from repeated dependency resolution checks.

Alternatively, in the non-structured approach (*ns-explore*), threads randomly select operations whose dependencies have been resolved. Each thread maintains a signal holder that asynchronously manages dependency resolutions for remaining operations. Specifically, when an operation O_i is successfully processed, the thread signals all other threads, which can then process operations dependent on O_i. Despite ensuring immediate resolution of operation dependencies and making more operations available, *ns-explore* can lead to higher message-passing overhead due to the need for countdown latch signal transmission along all directed edges.

B. Scheduling Unit Granularities

This dimension pertains to the task size that is scheduled, which includes single-operation (*f-schedule*) and multi-operation (*c-schedule*) scheduling. In *f-schedule* mode, threads manage individual operations, boosting scalability by maximizing parallelism and enabling immediate dependency resolution, as shown in Figure 7a. However, *f-schedule* suffers from high context-switching overhead. In contrast, *c-schedule* schedules batches of operations, reducing context-switching overhead but potentially delaying dependency resolution and impacting scalability, as seen in Figure 7b.

A key challenge with *c-schedule* is the potential for circular dependencies between coarse-grained scheduling units. As shown in Figure 7, O_3 depends on O_1, and O_5 depends on O_3, creating circular dependencies. To resolve this, *MorphStream* consolidates interdependent units into a single scheduling entity, prioritizing independent operations. This approach addresses circular dependencies but may reduce parallelism. Thus, the choice between *f-schedule* and *c-schedule* in *MorphStream* depends on the presence of cyclic dependencies and their efficient handling.

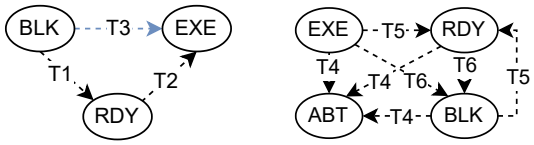
C. Abort Handling Mechanisms

This dimension decides how *MorphStream* handles transaction aborts, either eager abort or lazy abort. In the eager approach (*e-abort*), threads immediately abort transactions upon failure, minimizing the impact on concurrent operations. The implementation of *e-abort* differs by exploration strategy: with structured exploration, *e-abort* aborts failed operations and their dependents, then rolls back and restarts from the affected stratum. In non-structured exploration, *e-abort* uses a coordinator (the first operation, or "head") to abort the entire transaction and direct threads to redo dependent operations. The lazy approach (*l-abort*) delays action on failed operations, addressing them after fully traversing the TPG. This method reduces context switches but may require extra checks and iterations, leading to inefficiency. Regardless of the approach, *MorphStream* ensures correct scheduling by rolling back states modified by aborted operations to the latest version, using timestamped physical copies of each state. These versions are maintained until the TPG is fully processed, supporting rollback and windowing queries in *MorphStream*.

D. Heuristics Decision Model

Given the NP-complete nature of the original task graph scheduling problem and the added complexity of our context, our proposed solution is a lightweight, heuristic-based decision model. Informed by comprehensive microbenchmark studies and theoretical analysis (Table III), it effectively guides the *MorphStream* in making scheduling decisions.

Model Inputs. The model operates on seven properties from the constructed TPG (Table IV), reflecting different workload characteristics. Vertex computational complexity corresponds to the complexity of the user-defined function in the associated state access operation (C). Vertex degree distribution reflects state access distribution of the operation (θ), implying some



(a) Regular execution. (b) Abort and rollback.

Fig. 8: Six cases of state transition flow of an operation.

states are more frequently accessed. The ratio of aborting vertexes maps to the ratio of operations that need aborting (a), which needs to be profiled and estimated. The number of LDs, TDs, PDs relates to the number of transactions arriving during the batch interval (T), transaction length (l), with PDs also influenced by state accesses per operation (r). Lastly, cyclic dependency represents the presence of cycles when *c-schedule* is adopted, influenced by θ, T, l, and r.

Decision Model. As per Figure 6, the model operates on three parallel dimensions at runtime. *I) Exploration Strategies:* *s-explore* is chosen when the number of all dependencies is high and vertex degree distribution is uniform. In other cases, *ns-explore* is chosen for more flexible dependency resolution. *II) Scheduling Unit Granularities:* The model selects *c-schedule* when there are no cyclic dependencies among operations, TDs number is high, and PDs number is low. For other cases, *f-schedule* is chosen for its better scalability. *III) Abort Handling Mechanisms:* *l-abort* is chosen when the computational complexity of vertices is low and the ratio of aborting vertices is high. Otherwise, *e-abort* is chosen for its minimal impact on other operations' execution.

VI. Execution

MorphStream leverages the combination of finite state machine annotations and multi-versioning state management to establish a comprehensive framework that supports a variety of execution contexts, including standard operation execution, non-deterministic, and window-based operations.

A. Finite State Machine

Finite state machine (FSM) annotations enhance the TPG, creating an advanced *stateful TPG* (S-TPG) to manage concurrent operations' lifecycles and ensure correct commit/abort transactions. Each vertex can be in one of four states: (1) Blocked (BLK) indicates the vertex is not ready for scheduling due to unresolved dependencies. (2) Ready (RDY) means the vertex is ready for scheduling as all dependencies are resolved. (3) Executed (EXE) signifies the vertex has been successfully processed. (4) Aborted (ABT) means the vertex was aborted due to failed processing of itself or its dependencies. *MorphStream* continuously tracks state transitions in the TPG to ensure correct scheduling.

B. Standard Operation Execution

Serial and speculative scheduling, alongside abort handling (Figure 8), leverage the S-TPG and multi-version state table to ensure precise state access.

In **serial scheduling**, as shown in Figure 8a, an operation transitions from the BLK state to the RDY state (**T1**) once all its dependent operations reach the EXE state, making it ready for scheduling. Upon entering the RDY state, the operation is scheduled for execution by retrieving the target states from the multi-versioning state table and applying user-defined functions among them. If successful, this leads the operation to transition to the EXE state (**T2**).

Speculative scheduling offers an enhancement to execution concurrency. Specifically, an operation in the BLK state can be speculatively scheduled, despite having unresolved dependencies (**T3**). If the targeted states are unavailable, the state table management strategy in *MorphStream* ensures that the operation can wait until its targeted versions of states become available, although this may introduce additional context-switching overhead.

The state transition during **abort handling** is depicted in Figure 8b. If an operation fails during processing or its logical dependencies transition to ABT, the operation's state changes to ABT from any other state (**T4**). As part of this process, *MorphStream* leverages its state table management strategy to clear the state access effects of write operations on the state table. This involves removing all state versions added after the versions affected by the aborted operations. Rather than simply rolling back all affected operations to the BLK state, the system proactively evaluates whether the operation is ready to execute immediately after rollback, potentially transitioning from EXE or BLK to RDY (**T5**). However, if the dependent operations revert to RDY or BLK, the current operation must also revert to BLK due to unresolved dependencies (**T6**).

C. Non-deterministic Operation Execution

Non-deterministic operations present unique challenges that necessitate additional support at the system level. Inferring read/write sets via reconnaissance or offline symbolic execution [29], [30], or employing deterministic optimistic concurrency control [18], [31] for transaction validation, may result in high abort rates and increased contention in highly concurrent environments.

MorphStream implements a hybrid execution strategy accommodating both deterministic and non-deterministic operations by maintaining unified dependencies among state operations. Non-deterministic operations, handled similarly to standard ones, are executed once they transition from BLK to RDY. During such executions, *MorphStream* calculates a key reflecting the current workload requirements. The states accessed during these operations are logged in the S-TPG, facilitating deterministic rollbacks if transactions are aborted. To maintain data integrity and consistency, *MorphStream* deletes the associated versions of these records during any rollback or abort-induced state transitions.

D. Window-based Operation Execution

Window-based operation execution is a crucial aspect of *MorphStream*'s strategy, especially for streaming window queries like sliding or tumbling windows. *MorphStream* leverages its multi-versioning state table to execute these operations efficiently without additional overhead. This approach eliminates the need for auxiliary data structures to

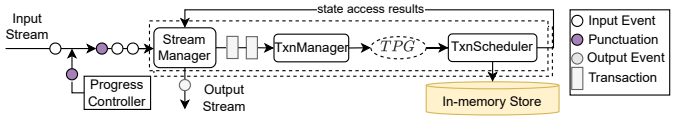


Fig. 9: The system architecture of *MorphStream*, the constructed TPG and shared state are stored in memory.

manage tuples across windows. Instead, *MorphStream* uses its multi-versioning state table, which maintains state versions in temporal order, allowing window tuples to be integrated as new state versions. The system also efficiently manages versions with ranged read/write capabilities, enabling rapid tuple retrieval through queries targeting specific state version ranges during window-based operations.

Window-based operations in *MorphStream* are triggered periodically based on configurations like window size and trigger settings. Each activation accesses a range of state versions and applies a user-defined aggregation function. This method is significantly enhanced by the state table's advanced querying capabilities, ensuring a seamless and efficient operation. This streamlined process not only boosts the system's performance but also enhances its ability to handle complex data processing tasks inherent in stream processing environments.

VII. Implementation Details

This section describes the architecture and implementation aspects of *MorphStream*.

A. Architectural Components and Interactions

As shown in Figure 9, the architecture of *MorphStream* comprises several crucial components that underpin its capabilities. The ProgressController plays an important role by assigning monotonically increasing timestamps to punctuation marks using a global counter. Punctuation-based mechanism (Section VII-B) ensures a consistent temporal perspective across all threads, which is essential for managing transaction dependencies effectively, as highlighted in Section IV. The StreamManager is responsible for the initial preprocessing and postprocessing of each input event.

TxnManager (TM), *TxnScheduler (TS)*, and *TxnExecutor (TE)* work together to manage the three phases of the execution paradigm: planning, scheduling, and execution. The TM is key to resolving transaction dependencies and constructing the *task precedence graph* (TPG). Upon receiving punctuation, TM updates the TPG to reflect current dependencies, preparing the framework for scheduling. The TS uses the updated TPG to make scheduling decisions, optimizing the parallel execution of operations. This process is central to *MorphStream*'s strategy for dynamically adjusting transaction processing to meet real-time demands. The TE executes operations scheduled by TS, managing state access and handling aborts via a finite state machine and multi-versioning state table. Each component plays a vital role in the transaction processing pipeline, ensuring seamless and effective operation from event preprocessing to transaction execution.

MorphStream preallocates shared mutable states in memory, dynamically expanding space for new states as needed. It employs a multi-versioning state table (Section VII-C) to support diverse operations in TSP applications.

B. Punctuation-based Mechanism

The punctuation-based mechanism synchronizes threads by using timestamped punctuation to coordinate stream and transaction processing. Punctuation markers ensure that all events after a punctuation have larger timestamps than all events before it, allowing the system to handle out-of-order arrivals confined within the boundaries of a punctuation. As described in Section IV, each incoming punctuation marker triggers the TxnManager to switch from the stream processing phase to the transaction processing phase to completely and efficiently capture dependencies among state transactions.

In addition, the punctuation-based mechanism also allows *MorphStream* to utilize punctuation intervals to control the number of state transactions in a batch. The batch size affects the number of logical dependencies, temporal dependencies, and parameter dependencies, which in conjunction with other factors (such as state access skewness) affects decisions on exploration strategies and scheduling unit granularities, as detailed in Section V. For example, *s-explore* is advantageous when the number of dependencies is high and state access distribution is uniform, while *ns-explore* is preferable in low punctuation interval and skewed workload for quick dependency resolution.

C. Multi-Versioning State Table Management

Multi-versioning state table management in *MorphStream* enables concurrent state access while optimizing window operations. By maintaining time-ordered version histories for each write, *MorphStream* ensures data consistency and high concurrency. This approach allows transactions to access specific state versions for window computations, read past versions without interference, and handle rollbacks efficiently. The framework consists of three components: version storage, index management, and garbage collection.

1) Version Storage: The version storage in *MorphStream* adopts an append-only strategy, where each write operation appends a new version of a tuple to the storage without modifying existing data. This structure minimizes contention and avoids the overhead of in-place updates, creating a complete history of all updates. For window-based operations, the append-only design ensures that historical versions of states are readily available for retrieval, enabling efficient processing of queries over sliding or session windows.

2) Index Management: To enable rapid version access, *MorphStream* employs a skip-list for index management. This structure provides logarithmic time complexity for locating versions, which is particularly beneficial for range-based queries in window operations. When executing a query that spans a specific time range, the skip-list allows *MorphStream* to bypass irrelevant entries, significantly reducing the latency of window-based computations. Additionally, the dynamic nature of the skip-list ensures compatibility with dependency tracking mechanisms, allowing efficient integration with the execution of triggered window queries.

3) Garbage Collection: In *MorphStream*, garbage collection is managed lazily, triggered only after the execution of a TPG. This ensures that expired state versions are cleared only after dependent operations are completed, preventing conflicts with active transactions. By deferring version removal, the system reduces garbage collection overhead, enhancing scalability. For window-based operations, this mechanism ensures that expired states are removed without affecting the performance of ongoing range queries or state updates.

Multi-versioning state table management optimizes window-based operations through these components. The temporal organization of state versions enables efficient range queries, while skip-list index allows quick access to relevant versions. The lazy garbage collection mechanism supports high performance by balancing cleanup with concurrent workload execution. This design enables *MorphStream* to efficiently handle dynamic workloads, such as session windows, while maintaining low latency and high throughput.

VIII. Evaluation

Our experiments were conducted on a dual-socket Intel Xeon Gold 6248R server with 384 GB of DRAM. Each socket has 24 cores at 3.00 GHz and 35.75 MB of L3 cache. To isolate NUMA effects, we used one socket and plan to explore NUMA further in future work [32]. For controlled testing, we pinned each thread to a single core and varied the number of cores from 1 to 24 to evaluate scalability. The server ran Linux kernel 4.15.0-118-generic and JDK 1.8.0_301, with heap sizes set at 300 GB for both initial and maximum configurations using the -Xmx and -Xms parameters. The G1GC garbage collector was used. Additionally, *MorphStream* was configured to retain temporal objects like processed TPGs and multiple state versions for a deeper analysis.

Our results first compare *MorphStream*'s performance with conventional SPEs and TSPEs under dynamic workloads. Next, we evaluate *MorphStream* against other TSPEs in scenarios involving windowed and non-deterministic state access. Finally, we demonstrate *MorphStream* 's effectiveness in supporting real-world use cases featuring these operations. Additional insights into workload characteristics, system overheads, and architectural profiling can be found in [9].

A. Performance Evaluation

1) Comparing to Conventional SPEs: In this section, we show that TSPEs significantly outperform conventional SPEs when handling TSP applications, using *Streaming Ledger* (SL) as an example. For conventional SPEs, we referenced the implementation proposed by Flink [4], which assumes transactional streaming jobs in SPEs are stateless while offloading shared state management to external systems such as Redis. We deployed SL on a standalone cluster with one TaskManager, 24 slots, and matched parallelism. To prevent out-of-memory errors, we allocated 100GB to the TaskManager heap. We note that without locking mechanisms (w/o Locks), Flink cannot ensure execution correctness. As shown in Figure 10, *MorphStream* significantly outperforms

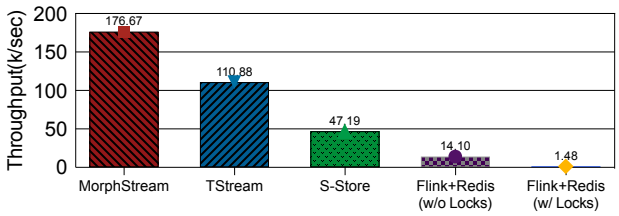
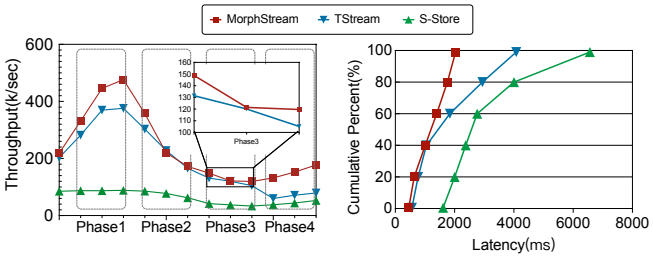


Fig. 10: Performance comparison among *MorphStream* and existing systems for running SL on 24 cores.



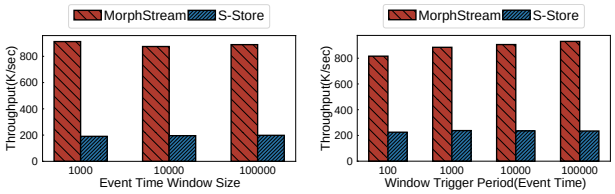
(a) Throughput (b) Latency

Fig. 11: Evaluation on dynamic workload.

TStream (1.6x) and S-Store (3.7x), and Flink (up to 117x). It is noteworthy that Flink, a popular conventional SPE, achieves orders of magnitude lower throughput in this application. Although disabling locks slightly improves Flink's throughput, it remains orders of magnitude lower than any of the TSPEs. Given this stark performance gap, we exclude Flink from further comparisons in the following discussions.

2) Evaluation on Dynamic Workloads: In this section, we show how *MorphStream* adapts to dynamic workloads by selecting optimal scheduling strategies, achieving consistently lower latency and higher throughput than state-of-the-art TSPEs. Using SL as the base application, we divide workloads into four phases, following the dynamic workload generation method proposed by Ding et al. [33]. Each phase is defined by varying trends, which tune workload parameters over time. For example, in a dynamic workload with an increasing tendency to abort transactions, we will increase the ratio of aborting transactions over time. Figure 11a and Figure 11b compare the throughput and latency of *MorphStream* against two state-of-the-art TSPEs: S-Store [8] and TStream [10]. We mark each phase in the dynamic workload using the dotted grey box.

In the first phase, SL includes numerous *deposit* transactions with scattered state access distribution, resulting in high LDs and TDs but few PDs. Due to uniform vertex degree distribution, *MorphStream* selects the *s-explore* strategy for dependency resolution and *c-schedule* for scheduling as guided by our decision model (Figure 6). Thus, *MorphStream* achieves up to 1.27x higher throughput than the second-best system. In the second phase, the workload gradually increases key skewness, concentrating dependencies on a small set of states. While all systems experience a performance drop, *MorphStream* dynamically shifts from *s-explore* to *ns-explore* for flexible dependency resolution, consistently outperforming S-Store. In the third phase, an increasing proportion of *transfer* transactions creates more dependencies between scheduling units. To mitigate this, *MorphStream* transitions from *c-schedule* to *f-schedule*, reducing dependency resolution overhead and maintaining stable throughput.



(a) Window Size (b) Window Trigger Period

Fig. 12: Evaluation of window queries.

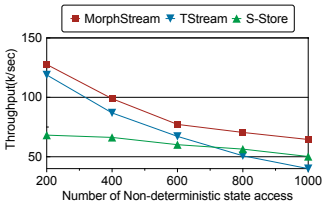


Fig. 13: Evaluation of non-deterministic operations.

There is no transaction abort in the first three phases, and the selection of the aborting mechanism in *MorphStream* does not matter. In the fourth phase, the workload simulates a rising ratio of aborting transactions. Initially, *MorphStream* employs the *e-abort* mechanism to eagerly abort failed operations but transitions to *l-abort* for batching aborts when aborts are frequent, minimizing context-switching overhead. While TStream struggles due to the overhead of reprocessing entire batches, *MorphStream* maintains stable performance, outperforming competitors by 2.2x to 3.4x.

A notable insight from Figure 11b is the significantly higher tail latency of TStream and S-Store compared to *MorphStream*. Their fixed scheduling strategies are optimized for specific workload conditions, causing inefficiency under changes like increased abort rates or skewness. By contrast, *MorphStream* dynamically adjusts its strategies to match workload variations, ensuring consistently lower latency.

B. Window and Non-deterministic State Access

We use the *GrepSum* (GS) benchmark [10], a typical stream application where large shared mutable states are frequently accessed and updated. In our adaptation, Grep processes each input event by initiating a state transaction to access table records, while Sum reads a list of states and writes the summation results back to the first one. We introduce periodic window reads and non-deterministic operations based on input and user-defined functions to meet our research objectives. By default, the benchmark includes 100,000 unique records (128 bytes each), with each transaction accessing two records.

1) Evaluation of Window-based Operations: We extended the GS benchmark to evaluate the performance of window-based operations by incorporating random state updates and periodic window readings. The application now handles two distinct types of state transactions: 1) Write-only transactions: Triggered by input events with update requests. 2) Read-only transactions: Initiated by input events involving window readings and sum aggregations. For consistency, experiments were conducted with an abort ratio of zero, a punctuation interval of 10,240 events, and window reads triggered every 100 write-only transactions. Each window read request accessed 100 random states within a default window size of 1,000, retrieving states up to 1,000 event-time units old. In the first experiment, we evaluated the impact of window size on throughput by varying the window size from 1,000 to 100,000 events. As shown in Figure 12a, larger window sizes increased state access overhead due to the retrieval of more state versions. Nevertheless, *MorphStream* effectively mitigated range read overhead using its efficient index management in multi-version storage, which only causes a slight throughput decline of approximately $\sim$5%. The second experiment evaluated the effect of window trigger periods, ranging from triggering every 100 to 100,000 events. As illustrated in Figure 12b, shorter trigger periods, corresponding to more frequent window queries, significantly reduced throughput, with a maximum decline of $\sim$15%. This drop was attributed to the increased computational overhead of frequent window-based operations, which are inherently more time-consuming. While *MorphStream* leverages multi-version storage to efficiently manage state access, frequent triggers amplify the demand for these operations, particularly in scenarios with high input rates.

We further compared *MorphStream* with S-Store under both experimental setups. TStream was excluded from this comparison as it does not naturally support window-based operations. Compared to S-Store, *MorphStream* achieved up to 4x higher throughput in window-based operations, surpassing the 3.4x improvement in non-window operations reported in [9]. This is because S-Store has to manage additional window states for historical input events, causing both storage and lookup overhead. Comparatively, *MorphStream* is more efficient in managing window state by leveraging its multi-version storage. These findings highlight *MorphStream* 's robustness and efficiency in handling window-based operations, showcasing its ability to maintain high throughput and low latency even under varying workload conditions.

2) Evaluation of Non-Deterministic Operations: We adapted the GS benchmark to evaluate non-deterministic operations, where state keys for writes are determined by both the item's value and a user-defined function. The experiment analyzed how varying the number of such operations impacts *MorphStream*'s performance. As illustrated in Figure 13, when the number of non-deterministic operations is low, *MorphStream* employs the *ns-explore* strategy for dependency resolution and *c-schedule* for scheduling, as guided by our decision model (Figure 6), enabling it to outperform other systems, particularly S-Store.

As the number of non-deterministic operations increases, throughput declines across all systems, but for different reasons. In S-Store, the impact is minimal since it processes dependent operations sequentially. In TStream, atomic operations that access the same state are scheduled together by threads. Non-deterministic operations increase dependencies between these scheduling units, limiting parallelism. Thanks to the adaptive scheduling mechanism, *MorphStream* switches to the *s-explore* strategy for dependency resolution and the *f-schedule* for scheduling, which efficiently handles a large number of dependencies. As a result, *MorphStream* mitigates performance degradation more effectively than both TStream and S-Store, leading to better overall performance.

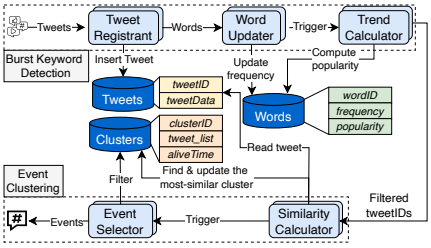


Fig. 14: Workflow of OSED.

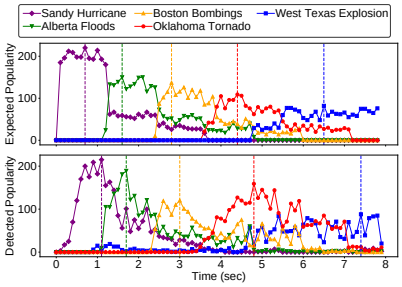


Fig. 15: Event popularity over time.

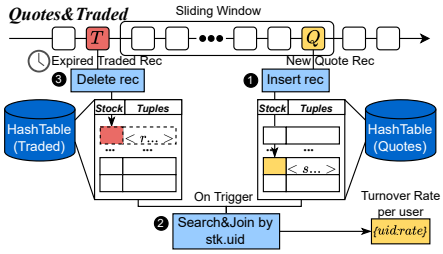


Fig. 16: Workflow of SEA.

C. Real-world Application Scenarios

This section shows *MorphStream*'s utility by implementing real-world use cases from Section II-B, both involving shared states with windowing or non-deterministic state access.

1) Online Social Event Detection (OSED): OSED is implemented as hybrid event detection [6], consisting of two stages: burst keyword detection and event clustering. As shown in Figure 14, the OSED pipeline consists of five streaming operators, all deployed over parallel executor threads for high performance. In the burst keyword detection stage, when new tweets arrive, Tweet Registrant threads insert them into the Tweet Table, decompose them into meaningful words, and distribute them downstream. Then, Word Updater threads atomically update the frequency of these words for each tweet. Once all words in the current window are updated, the Trend Calculator threads calculate the popularity of each word, identify burst keywords in the Word Table, and emit the corresponding tweets for clustering. In the second stage, the Similarity Calculator finds the most similar cluster for filtered tweets in the Cluster Table using cosine similarity. Once all tweets in the current window are clustered, the Event Selector identifies clusters with high growth rates as popular events.

MorphStream manages the Word Table and Cluster Table as shared, mutable states due to concurrent access by parallel threads. Additionally, to avoid the overhead of duplicating tweet data across the workflow (e.g., from the Tweet Registrant to the Similarity Calculator), the Tweet Table is also treated as a shared, mutable state. To maintain state consistency during such concurrent accesses, *MorphStream* encapsulates an atomic set of state access operations within a single transaction. Specifically, the Tweet Registrant performs single-key write transactions on the Tweet Table. The Word Updater manages multi-key write transactions on the Word Table, while the Trend Calculator executes single-key write windowing transactions on the same table. The Similarity Calculator conducts multi-key transactions, reading multiple keys from the Tweet Table and Cluster Table and writing the computed most similar cluster. Finally, the Event Selector processes single-key read windowing transactions on Cluster Table.

We conducted our analysis using a dataset of real-world tweets [34], comprising English tweets from five crisis events that occurred in the United States between 2012 and 2013. This dataset consists of approximately 30,000 tweets, both event-related and non-event-related. For our application, we deployed four executors for each operator, with a total punctuation interval of 400 tweets. Each thread was allocated 100 tweets per batch to execute. In Figure 15, we present the performance results of our OSED implemented using *MorphStream*, compared to the actual evolution of events over time. The popularity of events is measured by the number of new tweets merged into a specific event cluster within each time window. The five events are color-coded. In both the expected and detected popularity line charts, the solid lines represent the changing popularity of each event over time (in seconds), with markers indicating local maxima and minima. The dashed vertical line marks the global maximum popularity for each event. The delay between the expected and actual popularity is measured as the horizontal distance between the dashed lines for each event. The results demonstrate that our online event detection, supported by *MorphStream*, accurately detects the emergence of events and effectively captures changes in event popularity trends within seconds. We also observed that *MorphStream* achieved an overall throughput of up to 1.3k tweets per second for processing and detecting events. These findings provide compelling evidence of *MorphStream*'s ability to support complex real-time transactional stream applications efficiently.

2) Real-time Stock Exchange Analysis (SEA): SEA [12] can be implemented using the hash-based window join algorithm, which maintains two hash tables, one for each type of record in the stream. When a tuple arrives from the Traded or Quote stream, SEA inserts it into the corresponding hash table using stock IDs as keys. Once a window is triggered, the algorithm probes the two shared hash tables to find and join matching tuples by user IDs and calculates each user's portfolio turnover rate across different stocks. However, as demonstrated in our previous experiments, existing SPEs like Apache Flink provide limited support for such shared state access with transactional semantics. In contrast, *MorphStream* allows the hash table structure to be intuitively mapped to a shared state, where insertion and probe operations are modeled as state transactions.

An overview of the stock analysis workflow implemented using the hash-based window join algorithm is shown in Figure 16. *MorphStream* maintains two hash tables $Index(Traded)$ and $Index(Quotes)$, as shared state for the combined stream $Traded$ and $Quotes$. The state key, k, represents the stock ID, while the value, $< r, ... >$, contains all tuples that have arrived within the current window. Each tuple includes key attributes such as the user ID uid, the amount of stock quoted/traded, the quote/traded price, and other transaction details. the entire workflow of the SEA can be summarized in three steps. 1) Tuple Insertion: When a new tuple, s arrives, *MorphStream* inserts it to the associated table

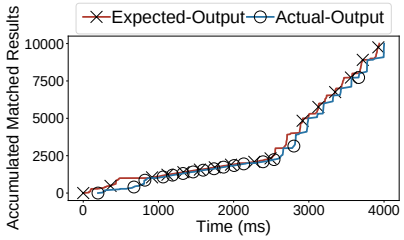


Fig. 17: Stock exchange accumulated matched results.

(either in $Quotes$ or $Traded$), which updates the multi-version state storage accordingly. 2) Window Join Execution: The window join operation is triggered periodically and follows these steps: a) Index Lookup: *MorphStream* searches for matching tuples by user id from both $HashTable(Traded)$ and $HashTable(Quotes)$ by efficient index lookup. b) Group Matching: Once matched tuples grouped by users are identified, e.g., $\{uid :< r, ... >\}$ and $\{uid :< s, ... >\}$, *MorphStream* calculates portfolio turnover rates for each user $uid : rate$. c) Result Propagation: The results $<user : rate>$ are propagated as the join output. 3) State Cleanup: Expired tuples from the expired window slide are deleted to maintain efficient state management.

Figure 17 shows the performance of the stock exchange analysis implemented on *MorphStream*. We used a real-world dataset [12] with tens of millions of quote and trade records. The application was deployed with 10 executors, and each executor processed 1k records per batch at a 1k batch interval. The focus of the evaluation was on the expected matched result generated by trade/quote events and the actual results output by *MorphStream*. The results show that *MorphStream* consistently outputs actual results within milliseconds. Additionally, *MorphStream* achieved a throughput of up to 70k events per second. These results validate that *MorphStream* efficiently handles real-time financial applications while ensuring ACID guarantees with high throughput and low latency.

IX. Related Work

Transactional stream processing engines (TSPEs) [4], [8], [11], [21], [10], [17] tackle complex scenarios where each input interacts with multiple keys, potentially causing conflicts in shared mutable states. This feature is becoming increasingly crucial for a wide range of streaming applications [7], [35] and emerging use cases [36], [4], [6]. In contrast to traditional SPEs, TSPEs like S-Store [8] and TStream [10] have explored techniques such as state partitioning and transaction decomposition to improve performance. However, these engines often rely on static scheduling, which struggles to accommodate dynamic workloads. In contrast, *MorphStream* employs an adaptive scheduling strategy that adapts in real-time based on workload characteristics, aiming to enhance throughput and reduce latency.

While TSPEs face unique challenges, especially in managing window-based operations [27], [37], [38] and handle non-deterministic operations, traditional database systems address transaction predictability and concurrency with different strategies. For window-based operations, recent advancements, including [39], [40], have explored optimizations in window maintenance and aggregate sharing. Schneider and Hirzel [39] reduce window maintenance overhead by leveraging efficient data structures and parallelizing window-based operations. Similarly, LightSaber [40] proposed a high-performance framework for window aggregation using *single instruction multiple data* (SIMD) and hardware acceleration. While these approaches optimize specific aspects of window processing in shared-nothing architectures, they do not address the broader transactional and concurrency challenges *MorphStream* solves. *MorphStream* integrates window maintenance with transactional guarantees, enabling efficient cross-key operations and dynamic scheduling that these optimizations do not address directly.

For non-deterministic operations, database systems use reconnaissance phases or offline symbolic execution to predict transaction needs [29], [30], [41], often resulting in high abort rates or heavy workload analysis. Some systems use deterministic optimistic concurrency control, which performs a validation phase in a fixed order without knowing the data set during execution. While it can improve performance, it may cause contention and higher abort rates in highly concurrent environments [18], [31]. In contrast, *MorphStream* offers a novel execution strategy for window-based and non-deterministic operations, extending TPG-based transaction execution to ensure efficient processing and adaptive scheduling in transactional stream processing.

MorphStream's choice of multicore processors is motivated by advancements in modern servers, which significantly increase computing power and memory capacity. For instance, recent scale-up servers can support hundreds of CPU cores and multi-terabyte memory [42], making them highly suitable for transactional streaming workloads. Shared-memory multicore architectures further enhance efficiency by simplifying data sharing and reducing communication overhead compared to distributed environments. Witnessing the emergence of modern commodity machines with massively parallel processors, *MorphStream* is specifically designed to capitalize on these strengths. Distributed environments are undeniably important for scaling TSPE. However, they have significant complexities, such as ensuring state consistency across nodes and optimizing cross-node communication. These challenges represent an exciting direction for future work as *MorphStream* evolves to meet broader scalability requirements.

X. Conclusion

In this work, we introduced *MorphStream*, a novel TSPE designed to optimize parallelism and performance for stream applications managing shared mutable states. Through a unique three-stage execution paradigm, *MorphStream* enables dynamic scheduling and parallel processing of data streams while handling shared state access conflicts. Building on this foundation, *MorphStream* is further enhanced with robust support for non-deterministic state access and window-based operations. Our experiment showcased the remarkable capabilities of *MorphStream*, significantly outperforming existing TSPEs in various scenarios.

ACKNOWLEDGMENT

We would like to express our gratitude to Mr. Siqi Xiang for his invaluable contributions to this work. We also thank Prof. Binbin Chen for his insightful comments, which greatly improved this paper. This research project is supported by the National Research Foundation, Singapore and Infocomm Media Development Authority under its Future Communications Research & Development Programme FCP-SUTD-RG-2022-005. Haikun Liu's work is supported by the National Natural Science Foundation of China under grant No.62332011. The corresponding author is Shuhao Zhang.

REFERENCES

[1] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Apache Storm," pp. 147–156, 2018. [Online]. Available: http://storm.apache.org/

[2] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink," 2018. [Online]. Available: http://flink.apache.org/

[3] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized Streams: Fault-Tolerant Streaming Computation at Scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 423–438.

[4] S. Ewen, *Serializable ACID Transactions on Streaming Data*, (2018). [Online]. Available: https://www.ververica.com/blog/serializable-acid-transactions-on-streaming-data

[5] I. Botan, P. M. Fischer, D. Kossmann, and N. Tatbul, "Transactional Stream Processing," in *Proceedings of the 15th International Conference on Extending Database Technology (EDBT)*, 2012, pp. 204–215.

[6] O. C. Sahin, P. Karagoz, and N. Tatbul, "Streaming Event Detection in Microblogs: Balancing Accuracy and Performance," in *International Conference on Web Engineering (ICWE)*, 2019, pp. 123–138.

[7] D. Wang, E. A. Rundensteiner, and R. T. Ellison III, "Active Complex Event Processing over Event Streams," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 4, no. 10, pp. 634–645, 2011.

[8] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, M. Stonebraker, K. Tufte, and H. Wang, "S-Store: Streaming Meets Transaction Processing," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 13, pp. 2134–2145, 2015.

[9] Y. Mao, J. Zhao, S. Zhang, H. Liu, and V. Markl, "MorphStream: Adaptive Scheduling for Scalable Transactional Stream Processing on Multicores," in *Proceedings of the 2023 International Conference on Management of Data (SIGMOD)*, 2023, pp. 1–26.

[10] S. Zhang, Y. Wu, F. Zhang, and B. He, "Towards Concurrent Stateful Stream Processing on Multicore Processors," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1537–1548.

[11] L. Affetti, A. Margara, and G. Cugola, "FlowDB: Integrating Stream Processing and Consistent State Management," in *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems (DEBS)*, 2017, pp. 134–145.

[12] SSE Team. (2018) Shanghai Stock Exchange, http://english.sse.com.cn/. Last Accessed: 2020-06-29.

[13] M. Fedoryszak, B. Frederick, V. Rajaram, and C. Zhong, "Real-Time Event Detection on Social Data Streams," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (SIGKDD)*, 2019, pp. 2774–2782.

[14] M. Hasan, M. A. Orgun, and R. Schwitter, "A Survey on Real-Time Event Detection from the Twitter Data Stream," *Journal of Information Science (JIS)*, vol. 44, no. 4, pp. 443–463, 2018.

[15] CFI Team. (2024) Portfolio Turnover Ratio in Stock Exchange, https://corporatefinanceinstitute.com/resources/career-map/sell-side/capital-markets/portfolio-turnover-ratio/.

[16] P. C. A. Katsifodimos, S. E. V. Markl, and S. H. K. Tzoumas, "Apache Flink™: Stream and Batch Processing in a Single Engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering (Bull. IEEE Comput. Soc. Tech. Commmon. Data Eng.)*, vol. 36, pp. 28–38, 2015.

[17] J. Zhao, H. Liu, S. Zhang, Z. Duan, X. Liao, H. Jin, and Y. Zhang, "Fast Parallel Recovery for Transactional Stream Processing on Multicores," in *Proceedings of 2024 IEEE 40th International Conference on Data Engineering (ICDE)*, 2024, pp. 1478–1491.

[18] Y. Lu, X. Yu, L. Cao, and S. Madden, "Aria: A Fast and Practical Deterministic OLTP Database," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 13, p. 2047–2060, 2020.

[19] Y. Liu, L. Su, V. Shah, Y. Zhou, and M. A. Vaz Salles, "Hybrid Deterministic and Nondeterministic Execution of Transactions in Actor Systems," in *Proceedings of the 2022 International Conference on Management of Data (SIGMOD)*, 2022, pp. 65–78.

[20] P. F. Silvestre, M. Fragkoulis, D. Spinellis, and A. Katsifodimos, "Clonos: Consistent Causal Recovery for Highly-Available Streaming Dataflows," in *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*, 2021, pp. 1637–1650.

[21] L. Affetti, A. Margara, and G. Cugola, "TSpoon: Transactions on a Stream Processor," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 140, pp. 65–79, 2020.

[22] S. Zhang, J. Soto, and V. Markl, "A Survey on Transactional Stream Processing," *The VLDB Journal (VLDBJ)*, vol. 33, pp. 451–479, 2024.

[23] A. Arasu, S. Babu, and J. Widom, "The CQL Continuous Query Language: Semantic Foundations and Query Execution," *The VLDB Journal (VLDBJ)*, vol. 15, pp. 121–142, 2006.

[24] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom, "STREAM: the Stanford Stream Data Manager (Demonstration Description)," in *Proceedings of the 2003 International Conference on Management of Data (SIGMOD)*, 2003, pp. 665–665.

[25] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras, "Exploiting Punctuation Semantics in Continuous Data Streams," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 15, pp. 555–568, 2003.

[26] L. Golab, K. G. Bijay, and M. T. Özsu, "On Concurrency Control in Sliding Window Queries over Data Streams," in *International Conference on Extending Database Technology (EDBT)*, 2006, pp. 608–626.

[27] J. Verwiebe, P. M. Grulich, J. Traub, and V. Markl, "Survey of Window Types for Aggregation in Stream Processing Systems," *The VLDB Journal (VLDBJ)*, vol. 32, pp. 985–1011, 2023.

[28] N. S. Nikolov and A. Tarassov, "Graph Layering by Promotion of Nodes," *Discrete Applied Mathematics*, vol. 154, pp. 848–860, 2006.

[29] A. Thomson and D. J. Abadi, "The Case for Determinism in Database Systems," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 3, pp. 70–80, 2010.

[30] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast Distributed Transactions for Partitioned Database Systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2012, pp. 1–12.

[31] Z.-Y. Dong, C.-Z. Tang, J.-C. Wang, Z.-G. Wang, H.-B. Chen, and B.-Y. Zang, "Optimistic Transaction Processing in Deterministic Database," *Journal of Computer Science and Technology*, vol. 35, pp. 382–394, 2020.

[32] S. Zhang, J. He, A. C. Zhou, and B. He, "BriskStream: Scaling Data Stream Processing on Shared-Memory Multicore Architectures," in *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*, 2019, pp. 705–722.

[33] B. Ding, S. Chaudhuri, J. Gehrke, and V. Narasayya, "DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 14, pp. 3376–3388, 2021.

[34] A. Olteanu, C. Castillo, F. Diaz, and S. Vieweg, "CrisisLex: A Lexicon for Collecting and Filtering Microblogged Communications in Crises," in *Proceedings of the AAAI Conference on Weblogs and Social Media (ICWSM '14)*, 2014, pp. 376–385.

[35] I. Botan, Y. Cho, R. Derakhshan, N. Dindar, L. M. Haas, K. Kim, C. Lee, G. Mundada, M.-C. Shan, N. Tatbul *et al.*, "Design and Implementation of the MaxStream Federated Stream Processing Architecture," *Technical Report/ETH Zurich, Department of Computer Science*, vol. 632, 2009.

[36] J. Meehan, C. Aslantas, S. Zdonik, N. Tatbul, and J. Du, "Data Ingestion for the Connected World," in *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2017, pp. 8–11.

[37] H. Zhang, X. Zeng, S. Zhang, X. Liu, M. Lu, and Z. Zheng, "Scalable Online Interval Join on Modern Multicore Processors in OpenMLDB," in *Proceedings of 2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 2023, pp. 3031–3042.

[38] J. Traub, P. M. Grulich, A. R. Cuéllar, S. Breß, A. Katsifodimos, T. Rabl, and V. Markl, "Scotty: General and Efficient Open-Source Window

Aggregation for Stream Processing Systems," *ACM Transactions on Database Systems (TODS)*, vol. 46, pp. 1–46, 2021.
[39] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu, "General Incremental Sliding-Window Aggregation," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, pp. 702–713, 2015.
[40] G. Theodorakis, A. Koliousis, P. Pietzuch, and H. Pirk, "LightSaber: Efficient Window Aggregation on Multi-Core Processors," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2020, pp. 2505–2521.
[41] S. Issa, M. Viegas, P. Raminhas, N. Machado, M. Matos, and P. Romano, "Exploiting Symbolic Execution to Accelerate Deterministic Databases," in *Proceedings of the 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, 2020, pp. 678–688.
[42] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron, "Scale-up vs Scale-out for Hadoop: Time to Rethink?" in *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC)*, 2013, pp. 20:1–20:13.

Jianjun Zhao received the bachelor's degree from North China Electric Power University, China, in 2020. He is currently working toward the Ph.D. degree in computer science and technology from Huazhong University of Science and Technology (HUST), China. His research interests include stream processing, transactional consistency, and parallel recovery, with a particular focus on scalable transactional stream processing on multicore processors and distributed environments.

Yancan Mao is a Research Fellow at the National University of Singapore since 2024. He received his PhD in Computer Science from the National University of Singapore in 2024, his Master's degree in 2019, and his Bachelor's degree from the University of Electronic Science and Technology of China in 2018. His research interests include distributed stream processing, multicore optimization, resource scheduling, and cloud-native data systems.

Zhonghao Yang received his bachelor's degree from Singapore University of Technology and Design (SUTD), Singapore, in 2022. He is currently working toward the Ph.D. degree in computing and data science from Nanyang Technological University (NTU), Singapore. His research interests included vector data management, stream processing, and parallel computing.

Haikun Liu (Member, IEEE) received the Ph.D. degree in computer science and technology from Huazhong University of Science and Technology (HUST), in 2012. He is a Professor with the School of Computer Science and Technology, HUST, China. He has co-authored more than 80 papers in prestigious conferences and journals. His research interests include in-memory computing, cloud computing, and distributed systems. He is a Senior Member of CCF.

Shuhao Zhang is a Professor in the School of Computer Science and Technology at Huazhong University of Science and Technology (HUST). Previously, he served as an Assistant Professor at the College of Computing and Data Science, Nanyang Technological University (NTU). From 2020 to 2021, he was a Postdoctoral Researcher with Prof. Volker Markl at Technische Universität Berlin. He earned his PhD from the National University of Singapore (NUS) and his Bachelor's degree from NTU. His research interests include parallel database systems, large language model (LLM) inference frameworks, and their integration with stream processing techniques.