The VLDB Journal (2024) 33:451–479
https://doi.org/10.1007/s00778-023-00814-z

REGULAR PAPER

A survey on transactional stream processing

Shuhao Zhang[1] · Juan Soto[2] · Volker Markl[2]

Received: 21 August 2022 / Revised: 27 August 2023 / Accepted: 1 September 2023 / Published online: 27 September 2023

Abstract
Transactional stream processing (TSP) strives to create a cohesive model that merges the advantages of both transactional and stream-oriented guarantees. Over the past decade, numerous endeavors have contributed to the evolution of TSP solutions, uncovering similarities and distinctions among them. Despite these advances, a universally accepted standard approach for integrating transactional functionality with stream processing remains to be established. Existing TSP solutions predominantly concentrate on specific application characteristics and involve complex design trade-offs. This survey intends to introduce TSP and present our perspective on its future progression. Our primary goals are twofold: to provide insights into the diverse TSP requirements and methodologies, and to inspire the design and development of groundbreaking TSP systems.

Keywords Transactions · Stream processing · Survey

1 Introduction

Stream processing, originating in the late 1990s and early 2000s with publish–subscribe systems and data stream management systems (DSMS) [10], has become vital for real-time data handling. Various frameworks like Apache Storm, Apache Flink, and Apache Kafka Streams, each with unique features, have been developed. The integration of relational queries with continuous stream processing has been explored since the first-generation stream processing engine (SPE) [37]. However, modern SPEs often lack the capability to maintain or query relational tables consistently during processing [2, 9, 49]. This leads to two key limitations: the absence of transactional guarantees and inconsistency across distributed systems [53].

Transactional stream processing (TSP) systems have emerged as a solution, blending real-time data stream management with ACID (atomicity, consistency, isolation, durability) guarantees [4, 7, 16, 30, 33, 39, 52, 76]. Unlike traditional databases, in TSP systems, transactions initiate via streaming events and can be triggered individually or in batches. To qualify as a TSP system, two core criteria must be met: (1) real-time data processing through immediate handling of discrete data tuples, thus negating the need for batch processing or extensive data storage; and (2) robust transactional integrity assured by ACID properties. Transactions in this context may alter the system's internal state—including data aggregates, intermediate results, or configurations—and potentially initiate external notifications or other side effects. Moreover, TSP systems provide interfaces that accommodate both continuous and relational queries, enabling versatile client interactions.

Integrating streaming and transactional capabilities introduces a unique set of challenges, resulting in varied design and implementation approaches [3, 16, 27, 36, 40, 59, 88]. These variations typically stem from application-specific requirements. TSP systems use streaming mechanisms to maintain up-to-date system states and offer real-time shared table views. Transactional features, on the other hand, ensure the consistent maintenance of these states and views.

The absence of standardized transaction models or query languages complicates the design of TSP interfaces and APIs. This lack of uniformity introduces a range of approaches, making it challenging to establish consistent terminology and feature sets across the field. Our survey aims to illuminate these variations and identify common threads. In TSP systems, query languages must support both continuous streams

✉ Shuhao Zhang
shuhao_zhang@sutd.edu.sg

Juan Soto
juan.soto@tu-berlin.de

Volker Markl
volker.markl@tu-berlin.de

[1] Singapore University of Technology and Design, Singapore, Singapore

[2] Technische Universität Berlin, Berlin, Germany

and transactional operations. The system's internal state not only is influenced by incoming streams, but can also be altered via explicit insert/update queries. This dual approach to state management adds complexity yet offers flexibility, allowing diverse transactional properties based on interaction modalities with the system.

1.1 Example use cases

We present two scenarios illustrating the value and necessity of transactional stream processing (TSP), with further application scenarios detailed in Sect. 5.

Leaderboard maintenance. Described by Meehan et al. [53], this use case pertains to real-time leaderboard updates during a TV voting show like American Idol (Fig. 1). Viewers vote for contestants, and the system must instantly and accurately rank them. The challenge includes maintaining different leaderboards (top-3, bottom-3, top-3 trending) and continuously updating them. Key requirements include ACID guarantees for validating and recording votes in a shared table, *Votes*, and ordering guarantees to tally votes sequentially.

Streaming ledger. Proposed by dataArtisans [33], the Streaming Ledger (SL) (Fig. 2) involves two types of requests accessing shared mutable states for fund transfers and deposits. The aim is to process a stream of these requests and output the results. To handle a large volume of concurrent requests, ACID guarantees and ordering guarantees are necessary. For instance, transfers may be rolled back if they violate data integrity, such as causing a negative account balance. Additionally, the system may need to process requests in temporal order by timestamp.

Modern SPEs and databases are inadequate for the described use cases. SPEs' restriction to disjoint state subsets [21] breaches transactional consistency for multi-key input tuples. Conversely, databases are ill-equipped for high-velocity input streams. TSP uniquely unifies ACID guarantees and stream processing.

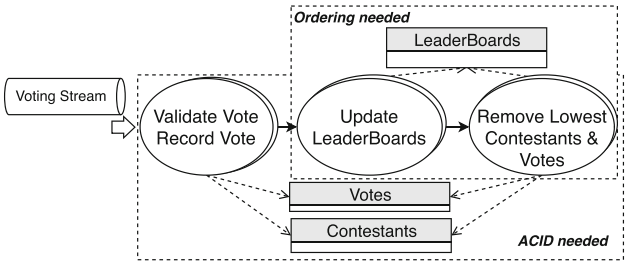


Fig. 1 Leaderboard Maintenance (*LM*) [53]

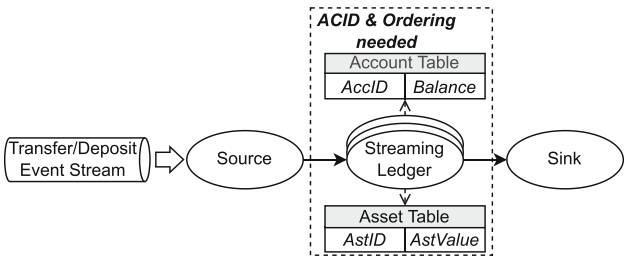


Fig. 2 Streaming Ledger (SL) [33]

1.2 Scope

This survey offers a comprehensive examination of TSP, encompassing the key challenges, design trade-offs, current trends, and potential future directions in the field. Our focus is on stateful stream processing, which involves on-the-fly processing of dynamic or streaming data, and is not confined to specific design and implementation choices such as dataflow engines and distributed systems. We emphasize six aspects of TSP:

- *Background.* The history, motivation, and various transaction models over data streams.
- *Properties.* Key properties like transactions, delivery guarantees, and state management.
- *Design aspects.* Examination of design factors including languages, APIs, and architectures.
- *Technologies.* Analysis of the technologies and alternatives used in TSP implementation.
- *Systems.* A review of representative TSP systems, highlighting features, strengths, and weaknesses.
- *Applications.* An overview of real-world applications and scenarios where TSP offers valuable insights and supports improved decision-making.

1.3 Outline of the survey

The remainder of this survey is organized as follows: Section 2 introduces TSP's background, conceptual framework, and transaction models over data streams. Section 3 discusses the taxonomy of TSP systems, including properties, design aspects, and implementation technologies. Section 4 surveys and compares early and recent TSP systems based on the taxonomy from Sect. 3. Section 5 shows applications and use cases such as stream processing optimization and concurrent stateful processing. Section 6 highlights open challenges and future directions including novel applications and hardware platforms. Section 7 concludes with a summary of key findings and insights.

2 Background

In this section, we provide an overview of key terms and concepts central to understanding transactional stream processing systems (TSP). We begin with fundamental definitions, present a conceptual framework of TSP, and explore various transaction models over data streams, referencing sources such as Babcock et al. [10].

2.1 Terms and definitions

A *data stream* represents a continuous flow of data reflecting underlying signals, such as network traffic streams. An *event* e is a 2-tuple $e =< t, v >$, comprising a timestamp and a payload, signifying the occurrence time and the relevant data, respectively. *Stream queries* consist of *operators*, fundamental computational units, that process events continuously. They can be traditional, like join and aggregation, or user-defined in modern stream processing engines (SPEs). *Windows* are subsets of streams, allowing the system to handle infinite streams, and they can be categorized into types like tumbling, sliding, and session windows. The *state* enables the juxtaposition of current data with historical data, vital for analyzing streams.

A *dataflow model* serves as a powerful abstraction in stream processing. A data flow breaks tasks into smaller units and coordinates their potentially parallel execution across nodes, typically represented as directed acyclic graphs (DAGs), where nodes are operators and edges denote data that is being transferred between the operators. Dataflow models are commonly used in stream processing. However, there are also other paradigms (e.g., relational model [39]), which we consider in this survey.

2.2 Conceptual framework of TSP

This section provides a comprehensive conceptual framework for transactional stream processing (TSP) systems, embracing various paradigms and models beyond the conventional dataflow approach. The expanded scope caters to a broader class of systems such as STREAM [39], recognizing the diversity and complexity in TSP systems.

2.2.1 Key components

A TSP system is comprised of five components: *transactions*, *transaction models*, *operators*, *scheduler*, and *storage*.

Transactions are critical components of TSP systems, ensuring that data is processed and maintained consistently and reliably. Transactions combine the real-time nature of stream processing with the reliability and consistency guarantees of traditional transactional systems. *Transaction models* describe the granularity and scope of transactions within a TSP system. In TSP, a transaction is considered committed when it has successfully completed an operation (e.g., insertion, update, deletion) and the system has confirmed that transactional changes are consistent with the desired transactional guarantees (e.g., consistency, isolation, durability). Upon committing a transaction, the system ensures that its effects are persistent and can be recovered in the event of a failure. This concept of "commit" is essential in TSP systems to maintain the integrity and consistency of the data during processing.

Operators are responsible for processing incoming and outgoing data, and performing operations, such as filtering, aggregation, transformation, or joins. Operators may also have mutable state that needs to be managed in a transactional manner. *Scheduler* manages the execution of operators and ensures that transactions are executed in the correct order, according to the chosen consistency and isolation models. Schedulers may need to handle out-of-order events, coordinate distributed execution, and manage resource allocation.

The *storage* of a TSP system handles both transient and permanent states. Due to performance reasons, TSPs will often employ two different memory subsystems for storing these different types of state: (1) a in-memory storage that holds the mutable state and intermediate results, allowing fast access but vulnerable to system failure; and (2) a persistent storage that ensures durability through databases or logs, which enable state recovery in failure scenarios.

2.2.2 Key design aspects

The conceptual framework of a TSP system covers the key aspects to consider when designing and implementing a TSP system, taking into account various classes and models, and including but not limited to dataflow models. This multi-dimensional view of the TSP system offers a rich and flexible understanding that accommodates different perspectives in the field and sets the boundaries for our analysis. The framework is inclusive and reflective of the dynamic and multifaceted nature of TSP systems.

Language. The language aspect of a TSP system defines the syntax and semantics for expressing streaming and on-demand queries, as well as the transactional properties (e.g., consistency, isolation, durability). This aspect is related to the transaction models and operators, as it provides the means for developers to define and manipulate transactions and the operations they perform.

Programming model. The programming model refers to the way developers interact with the TSP system to define and manage stateful operations and transactional guarantees. This aspect is related to transactions, transaction models, and operators, as the programming model provides the frame-

work for working with these components in a structured and organized manner.

Execution model. The execution model focuses on how the TSP system processes both streaming and on-demand queries while providing transactional guarantees. This aspect is related to the scheduler and operators, as the execution model determines how the scheduler manages the execution of operators and ensures that transactions are executed in the correct order, according to their consistency and isolation models.

Architecture. The *architecture* aspect of a TSP system encompasses the overall design and structural organization of the system. This includes the arrangement and interaction of the key components such as transactions, transaction models, operators, scheduler, and storage, and how they work together to meet the requirements of stream processing and data management. Specifically, the architecture: (1) defines the *physical layout* of the system, including the distribution and placement of processing nodes, network topology, and data storage locations; (2) dictates the *system's behavior*, including the strategies for parallel processing, fault tolerance, scalability, and resource management; and (3) shapes the *system's extensibility*, including how new features, components, or optimizations can be added or modified to meet evolving needs.

2.3 Transaction models over data streams

Next, we discuss some notable transaction models over data streams, along with their implementation approaches, to provide an understanding of transactional guarantees in stream processing.

2.3.1 Abstract models

Various transaction models have been explored for stream processing applications to address consistency guarantees. These models serve as processing paradigms in TSP systems, defining the boundaries of a transaction, i.e., the set of related state changes that are committed as a single unit. The state changes within a transaction can include both internal state modifications and external side effects, such as sending a message to a sink. Below, we describe five common transaction models in TSP systems:

Per-tuple transactions. In this model, each tuple in the data stream triggers a set of related state changes, and these changes are grouped and treated as a single transaction, adhering to ACID properties. Essentially, every tuple results in a transaction that encapsulates all state changes caused by that tuple. This approach is suitable for scenarios requiring atomic and isolated processing of individual events. However, it may introduce significant overhead due to frequent coordination between processing nodes.

Micro-batch transactions. Here, data streams are divided into small, bounded micro-batches, and transactions are executed over these batches. Unlike per-tuple transactions where each tuple defines a transaction, micro-batch transactions treat a whole batch as a single transaction boundary, grouping the changes caused by all tuples in the batch. This reduces the overhead associated with per-tuple transactions and allows for parallelism and optimization opportunities, but may introduce additional latency.

Window-based transactions. These transactions are executed over windows defined by criteria such as a unit of time or the number of events. The windows aggregate related events, and transactions are executed over these windows, encompassing all state changes within the window boundary. This approach provides stronger consistency guarantees, but can be challenging to manage, especially when dealing with out-of-order events or evolving window types.

Group-based transactions. Unlike window-based transactions that are defined by temporal or numerical boundaries, group-based transactions are executed over groups of related events defined by specific criteria, such as thematic relationships or business rules. This model provides fine-grained control over transaction boundaries, offering stronger consistency guarantees for complex processing tasks, but can be complex to manage.

Adaptive transactions. This model enables flexibility in defining transaction boundaries within TSP systems, allowing adjustments based on workload, system state, or application-specific needs. Rather than being an engine implementation detail, this adaptiveness is a fundamental part of the transaction model, providing a responsive framework tailored to the dynamic nature of stream processing. Implementing adaptive transactions can be challenging, as it requires real-time monitoring and the adaptation of transaction boundaries.

2.3.2 Implementation approaches

Implementation approaches for the aforementioned transaction models can be classified into three categories: unified transactions, embedded transactions, and state transactions. Depending on an application's specific requirements, an appropriate combination of implementation approach and particular transaction model should be chosen to achieve the desired performance, scalability, and fault tolerance capability.

Unified transactions. This approach embeds stream processing operations into a transaction, providing a single

framework for handling both stream processing and transactions. Unified transactions can potentially support various transaction models, as it allows flexibility in defining the scope and granularity of transactions. However, it might be more suitable for fine-grained transaction models, such as per-tuple or micro-batch transactions.

Embedded transactions. This approach embeds transaction processing into stream processing, allowing for transactional semantics without the need for separate transaction management. Embedded transactions can be more efficient for certain transaction models, particularly when a lightweight transaction mechanism is required. It might be better suited for per-tuple, micro-batch, or adaptive transactions, where the overhead of separate transaction management can be minimized.

State transactions. This approach separates transaction processing and stream processing, focusing on managing shared mutable state through transactions. State transactions can also support different transaction models, but are more suitable for scenarios where state management is a primary concern, such as window-based, group-based, or adaptive transactions.

3 Taxonomy of TSP

In this section, we examine a taxonomy of transactional stream processing (TSP) as illustrated in Fig. 3. The taxonomy is structured into three key categories: *Properties of TSP:* Here we examine the characteristics and requirements of TSP systems, including ordering, ACID properties, state management, and reliability. Analyzing these properties enables us to better understand the fundamental issues and challenges prevalent in TSP systems that seek to ensure accurate and reliable data processing. *Design Aspects of TSP:* Here we explore the design considerations in TSP systems, spanning transaction implementation, boundaries, execution, delivery guarantees, and state management. Investigating these design aspects aids us in evaluating the suitability of TSP systems for specific applications and provides insights into various design choices and trade-offs. *Implementation Details:* Here we address the practical aspects of TSP system implementation, such as programming languages, APIs, system architectures, and component integration. This section also discusses performance metrics and evaluation criteria for TSP systems. By examining these implementation details, we gain a deeper understanding of the practical challenges and proposed solutions in the development of TSP systems. Ultimately, this enables us to be more informed about the choices that must be made when designing or selecting which TSP system to employ for a given use case.

3.1 Properties of TSP

A TSP model is required to meet both the ordering properties of streaming operators and events, and the ACID properties of transactions, while also addressing state management, reliability, fault tolerance, and durability. These elements along with the CAP theorem's implications are crucial to the design and functionality of TSP systems. Hence, we will delve into these properties in the subsequent sections, starting with ordering properties followed by ACID properties.

3.1.1 Ordering properties

In TSP systems, there are two critical ordering properties: event ordering and operation ordering.

Event ordering. Event ordering requires that transactions be processed according to the order of their triggering events, typically based on timestamps or some other logical ordering mechanism. This property is crucial to ensure transactions are executed in the correct sequence, so as to avoid inconsistencies. It also helps prevent race conditions or out-of-order processing, which can occur in distributed TSP systems with high levels of parallelism. It is worth noting that the ordering schedule is determined explicitly by the input event rather than the transaction execution order.

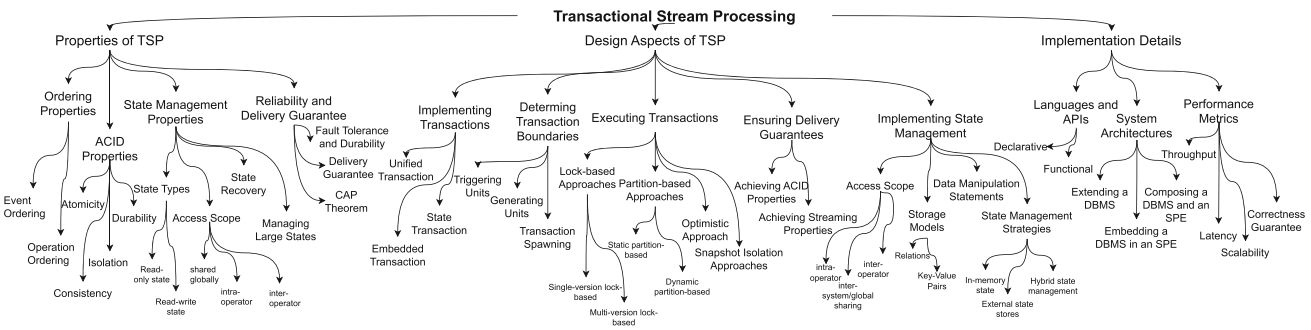


Fig. 3 Taxonomy of TSP

To maintain event order TSP systems may employ strategies such as locking, versioning, or optimistic concurrency control. Golab et al. [36] propose two stronger serialization properties with ordering guarantees. The first is called *window-serializable*, which requires a read-only transaction to perform a read either strictly before a window is updated or after all subwindows of the window are updated. The second is called *latest window-serializable*, which only allows a read on the latest version of the window, i.e., after the window has been completely updated. Instead of imposing an event ordering, FlowDB [3] enables developers to optionally ensure that the effects of transactions are the same as if they were executed sequentially (i.e., in the same order in which they started).

Operation ordering. In many TSP systems, applications can be represented using dataflow models, such as directed acyclic graphs (DAGs), where operators are connected by data streams [88]. Operation ordering refers to the sequence in which operators are executed in a TSP system, which impacts the correctness and efficiency of the system. Via operation ordering, the data that flows through the pipeline will be processed correctly, thereby contributing to the consistency and correctness prevalent in a TSP system. When an application is represented using a DAG, the operation ordering is determined by the directed edges between the operators. Although the operation ordering may be expressed differently for alternative representations, the ordering property guarantees the outcome will be the same. Consider a scenario where a TSP system adopts a dataflow model for stream processing—the ordering property is inherently upheld in such a setup. Nevertheless, if a database system incorporates TSP, an added prerequisite for operation ordering becomes necessary to guarantee consistency and accuracy, as detailed in the work of S-Store [53].

3.1.2 ACID properties

TSP systems manage the flow of data tuples, where transactions can involve state changes within the system and potential side effects, such as notifications to external components. The state in TSP refers to the information held within the system at any point, including data aggregates, intermediate results, or configurations. The application of a transaction in this context means executing a series of operations that may update this state, following certain rules or models. TSP can offer traditional ACID guarantees [12] similar to those in relational databases, with some necessary adaptations. These are briefly described below:

Atomicity. Atomicity ensures that transactions are either fully completed or aborted. In TSP systems, all operations within a transaction are either successfully processed together or not processed at all, thereby preventing partial updates that could lead to an inconsistent state. Atomicity in TSP varies depending on the transaction model. Traditional commit protocols, such as two-phase commit (2PC), ensure atomicity by coordinating commit or abort decisions among distributed participants. In contrast, sagas [35] allow the exposure of intermediate (uncommitted) state and require developers to define compensating actions for each operation, thereby offering a more flexible way to handle atomicity at the cost of strong isolation guarantees. Some TSP systems, like the one proposed by Wang et al. [84], relax atomicity in certain contexts, which enables developers to choose the desired consistency level. In such cases, alternative atomicity models, like sagas, can be adopted to balance the trade-offs between consistency, performance, and availability. Understanding these differences is crucial for designing TSP systems with appropriate atomicity guarantees.

Consistency. Consistency in the context of ACID refers to the requirement that every transaction moves the system from one consistent state to another. This involves the preservation of integrity constraints, which are rules defining the valid states of the data within the stream processing system, such as relationships between entities or domain-specific rules. In TSP systems, consistency also encompasses the processing and updating of data according to a specified consistency model, such as strong or causal consistency. This aspect is crucial in stream processing, where real-time data interactions guide the interactions between continuous transactions. The choice of consistency level may have implications for complexity and performance, as stronger consistency guarantees typically require more stringent enforcement of integrity constraints and other rules [3, 88].

Isolation. Isolation prevents concurrent transactions from interfering with one another [83]. In TSP systems, isolation is essential for ensuring that the output data remains consistent despite the concurrent execution of transactions triggered by the processing of concurrent input events. Different isolation levels can be provided by TSP systems, such as serializability, snapshot isolation, or read committed [3]. Some TSP systems may offer configurable isolation levels, allowing developers to adjust the isolation guarantees according to the application's specific needs.

*Durability.*Durability in TSP systems guarantees that once a transaction is committed, its changes are permanently stored, typically ensured through replication, logging, or checkpointing [83]. Unlike classic transactional systems, TSP's recovery mechanism might replay input streams to rebuild the state, a process that may not always restore an identical state due to factors such as concurrent processing and timing differences. The consequences of not reaching an identical state can lead to deviations in processing results, differentiating TSP from traditional transactional systems. Therefore, TSP systems need to satisfy properties like input preservation,

state maintenance, and output persistence, often achieved through strategies like replication, logging, or checkpointing. These strategies must balance trade-offs between performance, availability, and the specific requirements of the application. This complex yet essential aspect of durability in TSP systems underscores the need for careful design to build resilient TSP systems that meet the expectations and needs of the applications and users.

3.1.3 State management properties

Effective state management is essential in TSP systems to ensure data consistency, support the stateful processing of operations, and failure recovery. Among the primary state management properties are state types, access scope, state recovery, and the management of large states.

State types. States in TSP systems can be characterized by the interaction between different components or actors that may read or write the state: (a) **Read-only state:** In some scenarios, the state of the system is considered read-only from the perspective of external clients, as they can only observe or query the content of the state. Internally, the system may update its state upon receiving new events from the input streams, but these updates are not accessible to the clients. (b) **Read–write state:** In other scenarios, the state within the TSP system itself is modifiable as stream events are processed. This type of state allows both reads and writes during the execution, requiring careful management of concurrency control, especially when multiple entities (e.g., threads or processing units) may concurrently modify the same state. This categorization highlights the complexity of state management in TSP systems, reflecting the different interactions and permissions regarding state access from various perspectives within and outside the system.

Access scope. Access scope in TSP systems defines the visibility and accessibility of state information and can influence both the implementation and the programming interface. From an implementation perspective, the access scope dictates whether the state is shared globally, partitioned across parallel processing instances, or limited to specific operators or groups of operators. Shared states may include structures like the index of an input stream or other user-defined data structures shared among *threads* of the same operator, *operators*, and *queries*. From the programming interface perspective, the access scope also dictates the types of queries that can be made on the state. Some systems may support on-demand queries of the state, with varying levels of expressivity. Queries may be restricted to single operators or tables, single partitions, or may even integrate data from multiple operators. The flexibility and restrictions in querying significantly impacts the programming model and influences aspects like performance, consistency, and fault tolerance. By addressing both these perspectives, access scope forms a critical aspect of state management in TSP systems.

State recovery. State recovery in TSP systems is integral to maintaining the integrity and continuity of transaction processing over data streams. In the event of a system failure, state recovery ensures that processing can resume without loss of consistency or reliability. This involves preserving ACID even during failures. The state recovery property in TSP systems must prioritize a balance between recovery speed and assurance that the restored state aligns with the pre-failure state. This reflects the unique challenge in TSP systems of managing continuous, real-time transactions where both prompt recovery and adherence to transactional principles are paramount.

Management of large states. Managing large states is a fundamental challenge in TSP systems, particularly with growing data volumes and transaction numbers. Several traditional techniques might be adapted to TSP, though they require further investigation and innovation: (1) *Data sharding and state partitioning:* These methods [32] could provide scalability, but need careful exploration within TSP; (2) *State compaction:* A promising strategy [61, 77] for reducing storage needs, yet its practical implementation in TSP is still unexplored; (3) *State checkpointing:* Introducing state checkpointing [11] to TSP is currently being researched, and it could enhance recovery and durability; (4) *State replication, eviction, and expiration strategies:* While widely applied, these techniques [45] remain largely unexplored in TSP, but may offer benefits for resource management and fault tolerance.

3.1.4 Reliability and delivery guarantees

Fault tolerance, durability, and deliverability are essential properties of TSP systems. These properties ensure that TSP systems can accurately process data and maintain their state, even in the face of failures, duplicate messages, and out-of-order events.

Delivery guarantees. Delivery guarantees in TSP systems define how input events are processed especially when failures, duplicate messages, or out-of-order events occur. Common delivery guarantees include at-most-once, at-least-once, and exactly-once processing. Exactly-once processing is often the most desirable for TSP systems, as it ensures that each event is processed precisely once, irrespective of issues during processing.

Fault tolerance. A fault-tolerant TSP system continues processing events and adhering to delivery guarantees despite failures. Recovery mechanisms, such as state replication, checkpointing, and log-based recovery, enable a system to minimize data loss and quickly recover. However, the ability

to guarantee exactly-once processing requires that recovery leads to a consistent state that reflects all committed transactions.

Durability. Durability involves persistent data storage to safeguard its availability for retrieval. This property is essential for upholding delivery guarantees. However, as noted in Sect. 3.1.2, recovery mechanisms may not always lead to the same state. The commitment to exactly-once processing implies that the system must ensure a consistent recovery that honors all processed and committed input events, rather than strictly restoring the exact previous state.

CAP theorem. The CAP theorem applies to distributed systems, stating that it is impossible to simultaneously achieve consistency (across replicas), availability, and partition tolerance. In the context of TSP systems, these principles must be balanced according to specific needs. Emphasizing consistency and partition tolerance may slow response times, whereas prioritizing availability can lead to faster responses but potentially stale data. It is vital to differentiate consistency in the CAP theorem from transactional consistency in ACID transactions, as they relate to distinct aspects of TSP systems.

Remark 1 (Beyond exactly-once guarantee) TSP systems may face unique challenges requiring more stringent delivery guarantees than the exactly-once guarantee typically found in many SPEs. Specifically, TSP systems must replay failed tuples in the exact timestamp sequence of their triggering input events and prevent duplicate message processing. This is crucial because results depend on the local state of an operator and the time ordering of input streams.

One approach to achieving this advanced level of delivery guarantee involves checkpointing or archiving each input event before processing and sequentially replaying them in case of failure [23]. While this method provides the desired guarantee, it incurs significant overhead, making it unsuitable for many TSP systems. Consequently, further research is required to identify more efficient mechanisms that can satisfy the delivery guarantee needs of TSP systems while minimizing performance overhead.

3.2 Design aspects of TSP

The design aspects of TSP systems focus on the critical components required to create a functional and efficient TSP system. These components include the implementation of transactions, determining transaction boundaries, executing transactions, delivering guarantees, and managing the state. Understanding these design aspects is crucial for evaluating the suitability of different TSP systems for specific application requirements and providing insights into various design choices and trade-offs.

3.2.1 Implementing transactions

To actualize a transaction model over data streams, three primary approaches have been proposed: embedding stream processing operations into transactions (i.e., *unified transactions*), embedding transaction processing into stream processing (i.e., *embedded transactions*), or combining transaction processing and stream processing (i.e., *state transactions*). The choice among these approaches depends on the specific requirements and constraints of the system, and they may interact with each other in complex ways. Below, we discuss each of these in turn:

Approach 1: Unified transactions. Unified transactions integrate stream processing and transaction processing within systems adopting a relational query model, treating data streams and relational data uniformly [7, 39, 55]. This approach leverages existing relational data management techniques for transactional consistency, enabling a unified framework that accommodates both stream processing and transactional aspects. Several studies have explored the implementation of continuous queries as sequences of one-time queries, which are executed as a result of data source modifications or periodic execution [16, 53, 57, 58]. These studies illustrate methods to unify stream processing and transaction processing, so as to enable the use of a single execution engine for both types of operations.

Definition 1 (Unified transaction) Let $DS = \{S_i, S_o, \ldots\}$ be a set of data sources and sinks, and DI be the data items in DS. A unified transaction T is represented by the pair $T = (O_i, \leq_T)$, where:

- O_i: A set of read (r) and write (w) operations on DS, each operation $o \in O_i$ is a function $o : DI \rightarrow DI$ that modifies or retrieves a specific data item $d \in DI$. This set represents the operations on the data items within the given data sources and sinks.
- $\leq_T$: A partial order satisfying:
  - New streaming events and continuous query executions are represented as write (w) and read (r) operations, respectively.
  - For all $op, oq \in O_i$ accessing the same data item, with at least one write (w), either $op \leq_T oq$ or $oq \leq_T op$.

Constraints: The operations in O_i must adhere to certain constraints depending on the specific requirements of the TSP system, such as maintaining consistency or availability, as defined by the system's design and the nature of the data items.

Design aspects. The design space for unified transactions consists of several aspects, including data modeling, query

execution, and transaction management. (1) *Data modeling:* Data streams are represented as time-varying relations, with both input and output streams treated as continuous relations that are updated with new events, each marked with a timestamp. This representation, where each stream corresponds to a single relation, enables the use of relational algebra and SQL-like queries, and simplifies integration with existing database systems. (2) *Query execution:* Queries over data streams are often continuous: They are evaluated over an unbounded sequence of input data. Unified transactions represent continuous queries as a sequence of onetime queries triggered by data source modifications or periodic execution. This approach allows for the reuse of existing query processing techniques from relational databases while ensuring the continuous nature of stream processing is maintained. (3) *Transaction management:* To ensure transactional consistency and correctness, the unified transactions approach relies on relational data management techniques, in particular, concurrency control and recovery mechanisms, such as two-phase locking and logging, to provide isolation, atomicity, and durability guarantees.

Pros and cons: The unified transactions approach simplifies system architecture and leverages well-established relational data management techniques.

However, this approach may introduce additional overhead due to the transformation of data streams into relational data, a process that can be computationally expensive for several reasons. First, the transformation requires mapping continuous, unbounded data streams into discrete, time-varying relations, involving time windowing, data discretization, and possible aggregation. These computations require additional processing resources and may induce latency. Second, unifying the data model between streams and relational data may necessitate complex schema matching and data type conversions, further contributing to computational overhead. Finally, maintaining consistency and transactional guarantees within this unified model may require additional locking, logging, or other concurrency control mechanisms, which can also increase the system's complexity and resource requirements. Thus, this approach may not be suitable for all TSP systems, particularly those with strict latency requirements or complex stream processing needs.

Approach 2: Embedded transactions. Embedded transactions integrate transaction processing into the stream processing pipeline [69], allowing real-time processing while maintaining consistency and reliability. Several studies, including those exploring incremental continuous query processing with isolation guarantees [69], have explored the embedded transactions approach. For instance, Shaikh et al [69] propose a model that treats each incoming data item as a part of the processing pipeline, allowing real-time processing while maintaining consistency and reliability guarantees.

Definition 2 (Embedded transaction) Let $S = \{s_1, s_2, \ldots, s_n\}$ be a data stream consisting of data items and $O = \{o_1, o_2, \ldots, o_m\}$ be processing operations that act on the data items, including join operations with relations. An embedded transaction treats incoming data as a continuous flow, where:

- Each data item $s_i \in S$ initiates a subset of operations $O_i \subseteq O$, where each operation $o \in O_i$ is a function $o : S \rightarrow S$ that modifies or retrieves a specific data item.
- The processing follows properties such as isolation, atomicity, and consistency, with constraints to ensure that the operations meet the specific requirements of the system.
- Mechanisms like snapshot isolation and optimistic concurrency control may be used to maintain consistency in the presence of concurrent updates.

Design aspects. Key aspects of embedded transactions involve processing events as they arrive, managing state information between processing steps, and incorporating data partitioning, replication, and checkpointing to achieve transactional guarantees. Next, we examine each of these aspects in turn. (1) *Event-driven processing:* The embedded transaction approach employs event-driven processing, where each event in the data stream triggers one or more stream processing operations. This allows for low-latency processing and maintains the temporal order of data streams. (2) *State management:* State management is crucial in the embedded approach since it enables the retention and manipulation of information between processing steps. State management varies across systems, with some using distributed storage systems or in-memory data structures for efficient state management. (3) *Transactional guarantees:* The embedded transaction approach provides transactional guarantees, such as consistency, isolation, and durability within the stream processing pipeline. To achieve these guarantees, they employ techniques, such as data partitioning, replication, and checkpointing.

Pros and cons: The embedded transaction approach offers benefits, such as low-latency processing and the native handling of complex stream processing tasks. Nevertheless, implementing transactional guarantees within the stream processing pipeline could necessitate substantial effort. Furthermore, this approach might not be optimal for applications necessitating integration with established relational databases or traditional transaction processing systems. Developers should carefully assess the requirements and constraints of their specific application to determine whether this approach is appropriate for their TSP system.

Approach 3: State transactions. The state transaction approach merges transaction processing and stream processing within a single system and handles state access operations

as transactions. This enables TSP systems to provide transactional guarantees while processing unbounded data streams and ensure correctness via transactional semantics and the modeling of state accesses as *state transactions*.

Definition 3 (State transaction) Let $S = \{s_1, s_2, \ldots, s_n\}$ be a data stream of incoming events, $R = \{r_1, r_2, \ldots, r_p\}$ be shared mutable states that represent different aspects of the system's state, and $O = \{o_1, o_2, \ldots, o_m\}$ be processing operations that may read from S, modify R, or both. A state transaction T_i is represented by the triplet (S_i, R_i, O_i), where:

- $S_i \subseteq S$ is a subset of the data stream that the transaction operates on.
- $R_i \subseteq R$ is a subset of the shared mutable states that may be affected by the transaction.
- $O_i \subseteq O$ is a subset of operations that are applied to the elements of S_i and may modify the states in R_i.
- All operations within the transaction share a timestamp ts, ensuring coordinated execution and consistency across the state.

Design aspects. State transactions focus on managing shared mutable states through transactions. Key aspects include managing state information between processing steps and decoupling transaction processing from stream processing. Design considerations include dataflow models, state access operations, coordination mechanisms, and fault tolerance and recovery. Next, we examine each of these design considerations in turn. (1) *Dataflow models:* The state transaction approach typically uses dataflow models to process data streams consisting of interconnected stateful operators. These operators process events, update state, and produce output events, which is amenable for parallel and distributed processing. (2) *State access operations:* State transactions treat state access operations as transactions, where each operation is associated with a unique timestamp. This enables complex processing tasks and transactional guarantees, like consistency, isolation, and durability. (3) *Coordination mechanisms:* The state transaction approach coordinates state transactions across the dataflow pipeline using mechanisms, such as two-phase commit protocols, timestamp-based ordering, or conflict resolution strategies, to maintain consistency and isolation. (4) *Fault tolerance and recovery:* The state transaction approach provides fault tolerance and recovery mechanisms, such as replication, checkpointing, and logging, to ensure durability and resilience against failures.

Pros and cons: The state transaction approach integrates transaction processing into the stream processing pipeline, handles complex processing tasks, and provides transactional guarantees. However, this approach may introduce additional complexity due to the use of coordination and fault tolerance mechanisms. Developers should carefully assess their application's requirements and weigh the benefits against the potential complexities introduced by this approach.

Remark 2 (Comparing among approaches) We illustrate the differences among the three approaches using a common example scenario. Consider a system that monitors traffic in a smart city. It continuously receives data from sensors at intersections to control traffic lights, update maps, and alert drivers.

Unified transactions implementation. This approach integrates continuous stream processing with transaction processing:

$$T = (w(SensorData), r(TrafficMap), r(LightControl), w(UpdatedMap), w(Alerts))$$

where w and r denote write and read operations, respectively. The sequence includes writing sensor data, reading traffic maps and light states, and writing updates.

Embedded transactions implementation. Each incoming sensor data piece is treated as part of a continuous flow:

$$s_i = (SensorReading),$$
$$O_i = (AnalyzeData, UpdateMap, ControlLights, SendAlerts)$$

Operations include analyzing data, updating the traffic map, controlling lights, and sending alerts.

State transactions implementation. This approach handles both sensor data and mutable states:

$$T_i = (\{SensorReading\}, \{TrafficMapState, LightControlState\}, \{AnalyzeData\ ,\ UpdateMap\ , ControlLights,\ SendAlerts\})$$

The process includes reading data, managing shared states (traffic map and lights), and performing operations like analysis, updates, and alerting.

Summary: *Unified Transactions* integrate streaming with transactional operations, representing a complex sequence of reads and writes. *Embedded Transactions* treat the continuous flow of sensor data, translating each reading into operations. *State Transactions* manage shared mutable states (traffic map and lights), encapsulating the entire process in a state transaction. Each approach offers unique advantages and trade-offs: Unified transactions are suited for complex queries; embedded transactions for real-time continuous processing; and state transactions for robust handling of shared states within a transactional framework.

3.2.2 Determining transaction boundaries

Determining transaction boundaries is an essential aspect of TSP system design, as it defines which operations are grouped into transactions. It turns out that establishing transaction boundaries over streams can be quite flexible, particularly for state transaction implementation. First, various conditions can initiate a transaction (i.e., *triggering unit*), such as per input event or per batch of events with a common timestamp. Second, different entities can generate a transaction (i.e., *generating unit*), such as per operator and per query. Furthermore, transactions themselves can spawn additional transactions. Let us examine each of these settings in turn.

Setting 1: Triggering units. TSP systems rely on incoming streaming events to initiate transactions. The granularity of transaction boundaries is defined by various types of triggering units, including time-based, batch event-based, single-event-based, and user-defined triggers. Next, we describe each of these in turn.

Time-based triggers. Time-based triggers refer to transactional models in which the transaction boundaries are determined by time intervals. These intervals can be either fixed or dynamically adjusted based on the application requirements or the characteristics of the data streams. It often assumes that events with a common timestamp are executed atomically. This approach is employed in both academic projects, such as STREAM [7, 39] and commercial products, like Coral8 [31]. Time-based triggers are suitable for the concurrent aggregation of sliding windows, the association of transaction boundaries with window boundaries, and the management of long-running queries with specified re-execution frequencies [30, 36, 59].

Batch event-based triggers. These transactions are triggered when a batch of events with a shared characteristic (e.g., common timestamp, originating from the same stream) are processed. Batch event-based triggers are used in DataCell [47, 48], S-Store [53], and Chen et al.'s cycle-based transaction model [27, 28].They can handle large volumes of data and provide consistent processing across multiple streams.

Single-event-based triggers. In this type, a transaction is triggered for each incoming event. Single-event-based triggers ensure fine-grained control and consistency on an event-by-event basis. They have been implemented in various systems, such as Aurora and Borealis [1, 2], ACEP [84], SPASS [64], and TStream [88]. This type of transaction is suitable for applications requiring strict consistency guarantees and low-latency processing. The reason for this suitability lies in the individual handling of each event as a separate transaction. This ensures that every event is processed in isolation, maintaining strong consistency. Furthermore, by initiating a transaction for every individual event, the system can quickly react to incoming data, thereby facilitating low-latency processing. This can be critical for real-time applications where swift response to each input is required.

User-defined triggers. In this type, users can define custom triggering conditions based on their specific application requirements, which offers flexibility when establishing transaction boundaries and declaring processing guarantees. Botan et al. [16] and Chen et al. [26] demonstrate the use of user-defined transactions in their respective systems.

Setting 2: Generating units. While triggering units determine "when" a transaction is created, generating units focus on "who" generates a transaction. Transactions can be generated by user clients directly or through continuous queries on a per-query or per-operator basis. Next, we describe each unit type in turn.

Query-based generator. These transactions group operations involved in the onetime execution of an entire query [7, 39]. Early stream processing engines (SPEs) employed query-based triggers for the interactive processing of both relational and streaming data, such as the STREAM project [7, 39]. This type simplifies the transactional model, making it easier to manage and understand. However, it may lack flexibility in some cases, where individual operators within a query need to have separate transactional boundaries or different isolation levels.

Operator-based triggers. In this type, each operator in a query generates its own transactions, such as S-Store [53] and TStream [88]. Operator-based triggers provide a finer level of granularity and flexibility compared to query-based triggers, which enables more precise control over shared states in streaming dataflow graphs. This can lead to better performance and resource utilization in certain scenarios. However, this increased flexibility may also result in potential conflicts or dilemmas, such as deadlocks and contention, which may require additional mechanisms to resolve [53, 88].

User-defined triggers. Some applications may require ad hoc transactional queries or user-driven transactions during stream processing [3, 4, 26]. User-defined transactions allow users to specify where consistency needs to be enforced and which consistency constraints are required, as demonstrated in the work of Affetti et al. [3, 4]. This type grants more control to the user, thereby allowing them to tailor transactional semantics to their specific needs. However, this flexibility can make system-level optimizations more challenging, as the transaction types are not known in advance. Additionally, users bear the responsibility of ensuring that the system is free of any dilemmas or conflicts [3, 4].

Setting 3: Transaction spawning. Transaction spawning, i.e., the ability of transactions to trigger and generate other transactions, is another way to determine transaction boundaries in TSP systems. This concept is particularly relevant in sys-

tems that require complex interactions and dependencies between different transactions. For example, in a service-oriented architecture, where functionalities are treated as services, requiring atomic execution of those transactions results in composite transactions. This approach allows for a more flexible and dynamic processing flow, accommodating continuous service executions and interactions. Transaction spawning consists of a non-empty set of services, some of which have continuous executions, and others may spawn new transactions. In the context of TSP, this provides the means to represent intricate processing logic and dependencies, aligning with specific user or application requirements [80–82].

3.2.3 Executing transactions

Executing transactions involves processing the operations within a transaction according to the defined transaction boundaries and ensuring that the system maintains the required ACID and streaming properties. Table 1 summarizes the execution mechanisms adopted by relevant systems. These can be classified into five approach types: *single-version lock-based*, *multi-version lock-based*, *static partition-based*, *dynamic partition-based*, and *optimistic*.

Lock-based approaches. Lock-based approaches ensure the correct execution of transactions by controlling access to shared resources. Using locks to protect shared states, these methods prevent concurrent access and maintain consistency. Lock-based approaches can be classified into two main types:

a) Single-version lock-based: These approaches utilize a single version of the shared state and apply locks to ensure proper transaction execution. The challenge lies in balancing synchronization and performance without causing excessive contention or delays in transaction processing. We discuss three notable examples of single-version lock-based approaches as follows.

In STREAM [39], synopses enable different operators to share common states. To guarantee that operators view the correct version of a state, the system must track the progress of each stub and present the appropriate view (i.e., a subset of tuples) to each stub. This is achieved through a local timestamp-based execution model with a global schedule that coordinates the successive execution of individual operators via time slot assignments. Batches of tuples with the same timestamp are executed atomically to ensure progress correctness, with a simple lock-based transactional processing mechanism implicitly involved.

An earlier study by Wang et al. [84] describes a strict two-phase locking (S2PL)-based algorithm that allows multiple state transactions to run concurrently while maintaining both ACID and streaming properties. Unlike the original S2PL [12] algorithm, Wang et al. [84] lock each transaction ahead of all query and rule processing. In this process, each transaction's timestamp is compared against a monotonically increasing counter to ensure that the transaction with the smallest timestamp always obtains a lock first, thereby guaranteeing access to the proper state sequence. Once lock acquisition is complete, the system increases the counter and allows the next transaction to proceed, regardless of whether the transaction was fully processed. To fulfill event ordering constraints, read or write locks are strictly invoked in their triggering event order. However, the locking mechanism must synchronize the execution for every single input event, which may negatively impact system performance.

Oyamada et al. [57] propose three pessimistic transaction execution algorithms: synchronous transaction sequence invocation (STSI), asynchronous transaction sequence invocation (ATSI), and order-preserving asynchronous transaction sequence invocation (OPATSI). STSI processes transactions triggered by event streams one at a time, in event-arrival order. ATSI removes the blocking behavior of STSI by asynchronously spawning new threads that wait for the transaction to complete. OPATSI extends ATSI through a priority queue to further guarantee the order of the results.

b) Multi-version lock-based: These approaches employ multiple versions of shared states and use locks to control access to different state versions. The main challenge is ensuring that the correct state version is accessed while avoiding outdated writes.

A notable example is Wang et al. [84], who propose an algorithm called LWM (Low-Water-Mark), which relies on the multi-versioning of shared states. LWM leverages a global synchronization primitive to guard the transaction processing sequence: Write operations must be performed monotonically in event order, but read operations are allowed to execute as long as they can read the correct version of the data (i.e., its timestamp is earlier than the LWM). The key differences between LWM and the traditional multi-version concurrency control (MVCC) scheme are twofold. First, MVCC aborts and then restarts a transaction when an outdated write occurs, while LWM ensures that writes are permitted strictly in their timestamp sequence, preventing outdated writes. Second, MVCC assumes that the timestamp of a transaction is system-generated upon receipt, whereas LWM sets the timestamp of a transaction to the triggering event. This distinction enables LWM to maintain a more event-driven approach to transaction management, better aligning with the streaming nature of TSP systems.

In summary, lock-based approaches to transaction execution in TSP systems offer various methods for managing access to shared resources and maintaining consistency. While single-version lock-based approaches focus on balancing synchronization and performance within a single shared state, multi-version lock-based approaches provide greater flexibility by managing multiple versions of shared states.

Both types of approaches present their own challenges and trade-offs.

Partition-based approaches. Partition-based approaches to transaction execution in TSP systems involve dividing the internal states or transactions into smaller units, which can then be executed in parallel or with reduced contention. These methods aim to improve performance while maintaining consistency and adhering to event order constraints. There are two primary types of partition-based approaches:

a) Static partition-based: These approaches divide the internal states of streaming applications into disjoint partitions and use partition-level locks to synchronize access. This approach is suitable for transactions that can be perfectly partitioned into disjoint groups.

For example, S-Store [53] splits the streaming application's internal states into multiple disjoint partitions. The computation on each subpartition is performed by a single thread. To guarantee state consistency, S-Store uses partition-level locks to synchronize access. However, state partitioning only performs well on transactions that can be perfectly partitioned into disjoint groups, given that acquiring partition-level locks on cross-partition states significantly impacts performance due to the overhead.

b) Dynamic partition-based: These approaches involve decomposing transactions into smaller steps and executing them in parallel to improve performance while ensuring serializability and meeting event order constraints (e.g., TStream [88] and MorphStream [50]).

The *sagas* model [35] allows a transaction to be split into several smaller steps, each of which executes as a transaction with an associated compensating transaction. Either all steps are executed or in a partial execution compensating transactions are executed for steps that are completed. Thus, isolation is relaxed in the original transaction and delegated to the individual steps. It exposes an intermediate (uncommitted) state and requires developers to define compensating actions. A similar idea of splitting transactions has been adopted in TSP systems such as TStream [88] and MorphStream [50], but does not expose uncommitted states and hence does not require compensating actions.

In particular, TStream [88] is a recently proposed TSP system that adopts transaction decomposition to improve stream transaction processing performance on modern multi-core processors. Despite the relaxed isolation properties, TStream ensures serializability, as all conflicting operations (being decomposed from the original transactions) are executed sequentially as determined by the event sequence. The successor of TStream [88], MorphStream, pushes the idea further and proposes cost-model-guided dynamic transaction decomposition and scheduling to further improve the system performance.

Optimistic approach. Optimistic approaches avoid locking resources by employing timestamps and conflict detection mechanisms to maintain transaction consistency at the desired isolation level, aborting and rescheduling transactions when necessary. These approaches handle transactions by predicting the order of events or by speculative execution to improve system performance. The challenge is to ensure that speculation is accurate and efficiently manages rollback or recovery when needed.

Golab et al. [36] present a scheduler targeting *window-serializable* properties, which optimistically executes window movements and utilizes serialization graph testing (SGT) to abort any *read-only* transactions causing read–write conflicts. A conflict-serializable schedule is achieved if the precedence graph remains acyclic. They also suggest reordering read operations within transactions to minimize the number of aborted transactions. FlowDB/TSpoon [3, 4] pro-

Table 1 Execution mechanisms of TSP systems

Works	Approach	Key notes
STREAM [39], Wang et al. [84], Oyamada et al. [57], FlowDB/TSpoon [3, 4]	Single-version lock-based	Each state is maintained with a single copy; concurrent access is regulated by exclusive locks with an event ordering guarantee
Wang et al. [84]	Multi-version lock-based	Each state is maintained with multiple copies; concurrent access is regulated by read–write locks with an event ordering guarantee
S-Store [53]	State partition-based	Pre-partition states into disjoint partitions, and regulate concurrent access to each partition, similar to the single-version lock-based approach
TStream [88], MorphStream [50]	Transaction partition-based	Dynamically partition and regroup state transactions to avoid conflicts
Golab et al. [36], FlowDB/TSpoon [3, 4]	Optimistic	Optimistically schedule concurrent state transactions and abort transactions if conflicts arise
Several prior works [1, 16, 27, 29, 38]	Snapshot isolation	Implement the semantics of database transactions to defined segments of data streams, thereby assuring snapshot isolation during the processing of these segments

poses an optimistic timestamp-based protocol that refrains from locking resources and instead uses timestamps to ensure transactions consistently read or update versions aligned with the desired isolation level. If this is not feasible, transactions are aborted and rescheduled for execution. This approach aims to minimize contention and improve performance by avoiding lock-based mechanisms while still maintaining the necessary consistency and isolation requirements.

Snapshot isolation approach. These approaches employ snapshot isolation to split a stream into a sequence of bounded chunks and apply database transaction semantics to each chunk. Processing a sequence of data chunks generates a sequence of state snapshots. By storing multiple versions of values as commit and delete timestamps, readers can access the latest version of a state, ensuring consistency and isolation among concurrent transactions.

A number of TSP systems employ snapshot isolation [1, 16, 27, 29, 38], aiming to split a stream into a sequence of bounded chunks and apply the semantics of a database transaction to each chunk. By putting the operation on a data chunk within a transaction boundary, a state snapshot is produced. In this way, processing a sequence of data chunks generates a sequence of state snapshots. For example, Götze and Sattler [38] present a snapshot isolation approach for TSP. Each state has multiple versions of values, each stored as a *commit timestamp*, *delete timestamp*, and *value*. Consequently, readers can access the latest version of a state using the commit and delete timestamps. This approach provides consistency and isolation among concurrent transactions while avoiding the need for locking mechanisms, which can improve system performance.

3.2.4 Ensuring delivery guarantees

In this subsection, we explore various design aspects of TSP systems that help ensure reliability and delivery guarantees. We discuss strategies for achieving ACID properties and streaming properties under failures and their implications on TSP system design. For a comprehensive survey on fault tolerance mechanisms in SPEs, refer to [78]. While modern SPEs usually offer fault tolerance mechanisms while ensuring various delivery guarantees, they may not always fulfill the requirements of TSP due to the combined need to satisfy ACID and streaming properties.

Achieving ACID*properties.* In the event of a failure, TSP systems generally need to recover all states, including input/output streams, operator states, and shared mutable states. This ensures committed transactions remain stable, while uncommitted transactions do not impact this state. Transactions that have started, but have not yet been committed should be undone upon failure and reinvoked with the correct input parameters once the system is stable again. This necessitates an upstream backup and an undo/redo mechanism akin to an ACID-compliant database.

For instance, TSP systems must guarantee *atomicity* when updating shared states, even under failures. An atomic transaction ensures a commit either fully completes the entire operation or, in cases of failure (e.g., system failures or transaction aborts), rolls back the database (or shared states in TSP) to its pre-commit state. Journaling or logging in database systems mainly accomplish atomicity, while distributed database systems require additional atomic commit protocols to ensure atomicity. Regrettably, most prior works on TSP either do not explicitly mention their mechanisms to ensure atomicity under failure [3, 88] or rely on mechanisms provided by their storage systems (e.g., traditional database systems [53]). Making this more transparent could help users better understand which properties are not guaranteed when employing a TSP system in practice.

Achieving streaming properties. To satisfy streaming properties further, the recovered states in TSP systems should be equivalent to the one under construction when no failure occurred. Achieving this requires an order-aware recovery mechanism [72]. However, the commonly adopted recovery operation in modern SPEs, particularly the parallel recovery operation, might result in different transactional states due to the absence of guarantees on the event processing sequence during recovery. To the best of our knowledge, there is still no in-depth study on designing efficient fault tolerance mechanisms for TSP systems.

3.2.5 Implementing state management

State management is a crucial aspect of transactional stream processing (TSP) systems, as it enables the coordination of concurrent transactions, maintains consistency, and provides fault tolerance [6, 88]. The design space for state management in TSP systems can be characterized by several dimensions, such as access scope, storage model, data manipulation statements, and state management strategies. Next, we delve into these dimensions and explore their implications for system design and optimization.

Access scope. The access scope of state management ranges from intra-operator to inter-systems. Depending on the application's requirements, TSP systems may need to manage state locally within a processing node or share state across multiple nodes or even external systems [21, 38]. It is worth noting that when OLTP workloads are implemented in a TSP system, the access scope of a shared state is within a transaction, which can be attributed to a single operator or multiple operators. *a) Intra-operator state management* focuses on maintaining state among instances of a single operator, making it suitable for applications with localized data access patterns and minimal coordination requirements

[36]. *b) Inter-operator state management* involves sharing state across multiple operators within the same query/system [3, 88]. This approach is particularly relevant for applications that require coordination among different operators. *c) Inter-system/global state management* extends the scope of state sharing even further, enabling TSP systems to exchange state information with external systems, such as other stream processors, databases, or distributed file systems [52]. This approach allows TSP systems to leverage the capabilities of external systems, such as query processing or storage, and can facilitate seamless integration with existing data processing pipelines. However, managing state across system boundaries can introduce additional complexity, latency, and potential consistency issues.

Storage models. There are two primary storage models for implementing state management in TSP systems: relations and key–value pairs. Each has its trade-offs and implications for system design and optimization.

a) Relations: In this model, states are represented as time-varying relations that map a time domain to a finite but unbounded bag of tuples adhering to a relational schema [39, 53]. This approach leverages well-developed techniques from relational databases, such as persistence and recovery mechanisms. Storing states as relations can help minimize system complexity, especially when a foreign key constraint is required in TSP [52]. However, incorporating time into the relational model can add complexity to query processing and optimization [39].

Representative examples include STREAM, S-Store, and TStream. STREAM [39] represents the state as a time-varying relation, mapping a time domain to a finite, but unbounded bag of tuples adhering to the relational schema. To treat relational and streaming data uniformly, STREAM introduces two operations: *To_Table* to convert streaming data to relational data and *To_Stream* to convert relational data to streaming data. S-Store [53] does not implement its own state management component, but instead relies on H-Store [74] to ensure the transactional properties of shared states represented as relations. TStream [88] uses the Cavalia relational database [86] to support the storage of shared states.

b) Key–Value Pairs: In this model, states are represented as key–value pairs, which simplifies the design of TSP systems [6, 21]. This approach is suitable for scenarios that mainly require select and update statements for manipulating shared states during stream processing [4, 50]. However, it may not be the best choice for applications that require more complex data manipulation operations or constraints, such as foreign key constraints [52].

Representative examples include MillWheel [6], Flink with RocksDB, AIM (Analytics in Motion) [17], FlowDBMS/TSpoon [3, 4], and TStream/MorphStream [50, 88]. MillWheel [6] maintains state as an opaque byte string on a per-key basis, with users implementing serialization and deserialization methods. The persistent state is backed by a replicated and highly available data store, such as Bigtable [25] or Spanner [32], ensuring data integrity and transparency for the end user. Flink [21] relies on an LSM-based key–value store engine called RocksDB [65] to support shared queryable state. Götze and Sattler [38] also adopt a key–value store for transactional state representation, using multi-version concurrency control, where each state (i.e., key) has multiple *commit timestamps*, *delete timestamps*, or *values*.

AIM [17] represents state in a distributed in-memory key–value store, where nodes store system state as horizontally partitioned data in a *ColumnMap* layout. The *Analytics Matrix* system state provides a materialized view of numerous aggregates for each individual customer (subscriber). When an event arrives in an SPE, the corresponding record in the *Analytics Matrix* is updated atomically. In FlowDBMS/TSpoon [3, 4], a key–value store is employed, with the state maintained by a special type of stateful stream operator called the *state operator*. All state access requests must be routed to and subsequently handled by state operators defined in the application.

Data manipulation statements. TSP systems need to define and support different data manipulation statements employed in applications that constrain both system design and potential optimizations. These statements may include operations such as insert, update, delete, and query, which must be executed efficiently and consistently in the context of transactional stream processing.

Storing shared states as relations could be a reasonable choice of system design when applications require *insert* (I) or *delete* (D) statements and need to maintain foreign key constraints, such as in streaming ETL [52]. However, when applications only need *select* (S) and *update* (U) statements for manipulating shared states during stream processing, storing shared states as vanilla key–value pairs is sufficient and simplifies the design of TSP systems. Specific optimizations should be adopted by the TSP systems according to application needs.

State management strategies. The choice of state management strategy can significantly impact system performance, fault tolerance, and scalability [6, 21, 74]. There are three main strategies for managing state in TSP systems: *a) In-memory state:* This strategy maintains state within the processing nodes' memory, enabling low-latency access. However, it can be limited by available memory and may require replication and distributed transactions for fault tolerance and consistency guarantees [17, 88]. *b) External state stores:* In this strategy, state is stored in external systems, such as transactional databases or distributed key–value stores

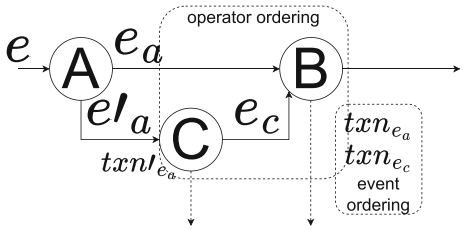


Fig. 4 Example of timestamp assignment dilemma

with transactional support [38, 65]. This approach allows for improved fault tolerance, consistency guarantees, and scalability, but may introduce additional latency [25, 32]. *c) Hybrid state management:* This approach combines the advantages of in-memory state and external state stores, using in-memory caching to minimize latency and external transactional storage for fault tolerance, consistency guarantees, and scalability [3, 4].

Remark 3 (Failure of concurrency control protocols) Conventional concurrency control (CC) protocols, widely used in OLTP database systems, fails to guarantee the properties of TSP Systems. To illustrate why, we use conventional timestamp-ordering concurrency control (T/O CC) as an example [13], with discussions also found in prior work [84]. Let $txn_1 = write(k1, v1)$ and $txn_2 = read(k1)$ be two distinct transactions. For simplicity, assume that only these two transactions are in the system, and that events are parallel processed. Suppose that txn_2 is generated and processed by the system earlier.

If $txn_2.ts > txn_1.ts$, then both transactions will be successfully committed. However, since txn_2 is processed earlier, it will read the old state value of $k1$, violating the event order constraint, leading to a serial order of $txn_2 \rightarrow txn_1$. On the other hand, if $txn_2.ts < txn_1.ts$, then txn_2 will be successfully committed, but txn_1 will be aborted since the writes come too late, making the undo of an externally visible output or action unacceptable in TSP applications.

Similarly, other conventional CC protocols may result in either the wrong serial order or the need to abort one of the transactions, leading to incorrect serial order upon restart. In other words, conventional CC protocols are not yet ready for such *event-driven* transaction execution.

Remark 4 (Timestamp assignment dilemma) In TSP systems, aligning transaction timestamps with the triggering events is a reflection of external consistency or linearizability in distributed systems. This alignment ensures proper ordering, but can lead to dilemmas. The dilemma, illustrated in Fig. 4, arises when transactions are generated by both external and internal events, such as operator outputs. Consider a scenario involving operators A, B, and C processing events and generating new transactions. Events are parallel processed, and txn_{e_a} and txn'_{e_a} can infinitely wait for each other to be committed, leading to a deadlock due to conflicting ordering constraints. Two potential solutions to this dilemma are: (1) enforcing additional ordering constraints between operators or (2) diversifying timestamps for generated events. However, these solutions present challenges, as implementing strong consistency like linearizability may affect latency, and a generalized solution to this dilemma remains unresolved.

3.3 Technologies employed in TSP implementation

This section delves into the practical aspects of implementing TSP systems, such as the choice of programming languages and APIs, system architectures, and the integration of various components to achieve a well-rounded TSP system. We also discuss performance metrics and evaluation criteria for TSP systems.

3.3.1 Languages and APIs

TSP systems should provide user-friendly and expressive languages that can easily define complex transactions, data manipulations, and processing logic over data streams while addressing aspects such as ordering properties, state management, and delivery guarantees. However, TSP systems do not yet have a standard transaction model or language, which complicates the selection of appropriate languages and APIs. Let us examine both declarative languages and functional languages in turn.

Declarative languages. The STREAM system [7, 39, 55] supports a declarative query language called CQL (Continuous Query Language), which is designed to handle both relational data and data streams. Coral8's [31] continuous computation language (CCL) has a SQL-like syntax and supports both data streams and event streams. Franklin et al. [34] introduced TruSQL, a relational stream query language that fully integrates stream processing into SQL, including persistence. Although declarative languages like SQL are indeed utilized in relational databases to manage transactions, TSP presents unique challenges such as preserving stream ordering property, which might not be directly addressed by conventional declarative languages. Extensions or modifications to these languages may be necessary to provide guarantees and facilitate correct interaction among stream queries in the context of TSP.

Functional languages. Functional languages [6], influenced by MapReduce-like APIs, are an alternative to declarative languages for expressing state abstractions and complex application logic than SQL-like declarative languages. Streaming systems like Flink embed functional/fluent APIs into general-purpose programming languages, allowing users

to define custom data flows akin to the Aurora system [2]. This design is also present in TSP systems, such as FlowDBMS and TStream [3, 4]. Functional languages offer greater flexibility and expressiveness, enabling custom processing logic and leveraging existing functional programming paradigms. In the context of expressing transactions, they provide fine-grained control, but may introduce additional complexity and a steeper learning curve. In contrast, declarative languages simplify query formulation, but may lack the nuances required for TSP-specific transaction handling. Further research is needed to explore these trade-offs.

3.3.2 System architectures

Given the properties and requirements discussed, TSP applications usually necessitate the use of SPEs in conjunction with data storage and analysis frameworks, such as database management systems (DBMSs), to create software architectures that integrate data storage, retrieval, and mining. Three approaches can be employed to construct a TSP system: (1) extending a DBMS, (2) embedding a DBMS in an SPE, and (3) composing a DBMS and an SPE. The choice of architectural approach for TSP systems depends on the specific requirements and constraints of the stream processing application. Each approach offers a unique set of trade-offs and considerations, making them more or less suitable for different use cases. Next, we explore each approach in turn.

Extending a DBMS. Extending a DBMS for TSP involves incorporating stream processing functionality into traditional DBMSs. By doing so, these systems aim to provide a unified platform for stream processing and traditional database management tasks while maintaining the strong consistency, isolation, and fault tolerance properties of traditional databases. Since state is managed directly within the DBMS, it can be shared across queries and operators, and provide durability and consistency. Since transactions are supported in the underlying DBMS, it offers strong consistency guarantees. This setup is particularly advantageous in applications, such as financial systems, where the integrity and correctness of the data are paramount.

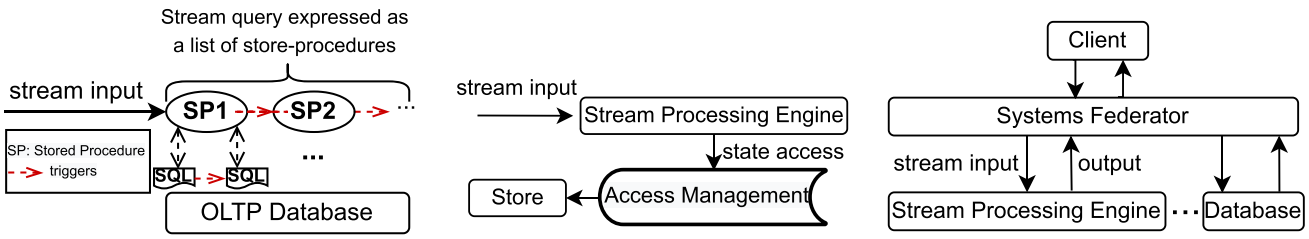
Notable examples of this approach include DataCell [48], MaxStream [15], Truviso Continuous Analytics system (TruCQ) [43], and S-Store [53]. While extending a DBMS for TSP offers a unified platform, it also has limitations, such as difficulty in efficiently supporting native or hybrid stream processing applications and challenges in handling real-time processing and stateful operations. Consequently, alternative approaches might be more suitable for specific TSP applications.

Advantages of this approach include: a) leveraging the existing features and infrastructure of a DBMS for transactional support, query processing, and data management; b) simplifying the system architecture by integrating stream processing functionality within the DBMS, reducing the need for additional components or interfaces; and c) potentially providing strong consistency guarantees by directly utilizing the transactional mechanisms of the underlying DBMS. Disadvantages include: a) potentially being less flexible and adaptable to the specific requirements of stream processing applications, as it inherits the architectural constraints of the underlying DBMS; and b) possibly having limited scalability and performance due to the constraints of the underlying DBMS, which may not be designed for high-velocity, high-volume data streams.

Embedding a DBMS in an SPE. The embedding approach involves integrating an SPE with an embedded key–value store or DBMS to manage state, provide transactional support, and handle storage capabilities. This method enables stream processing systems to take advantage of the features of the embedded DBMS while maintaining the flexibility and efficiency of stream processing. In this approach, state management occurs within the embedded DBMS, which can also be shared across queries and operators. However, the sharing model depends on the specific integration between the SPE and embedded DBMS. Durability can be achieved via the underlying DBMS.

Examples of this approach include Aurora [2] and its successor Borealis [1], and TStream [88] and its successor MorphStream [50]. Embedding a DBMS within an SPE allows for various transactional models over streams, but also introduces challenges such as integration complexity and potential performance bottlenecks or resource contention issues. This approach is better suited for applications that require a balance between the performance and scalability of stream processing and the transactional support and data management capabilities of a DBMS. It can be particularly useful when the stream processing workload is dynamic and demands efficient state management and transactional support. Examples of such applications include real-time analytics, social network analysis, and large-scale data processing tasks like log analysis or clickstream processing.

Advantages of this approach include: a) enabling TSP systems to benefit from the features of both the SPE and the embedded DBMS, combining their strengths and providing a unified platform for stream processing; and b) offering improved performance and scalability by facilitating concurrent data processing and state management, as well as leveraging the distributed nature of modern SPEs. Disadvantages of this approach include: a) careful integration of the embedded DBMS with the SPE may increase complexity and impact overall system latency; and b) the performance of the embedded DBMS might be influenced by the stream processing workload, potentially causing bottlenecks or resource



(a) Extending a DBMS (b) Embedding a DBMS in a SPE (c) Composing a DBMS and an SPE

Fig. 5 Illustrations of alternative system architectures of TSP systems

contention issues. These problems can be mitigated by carefully tuning the embedded DBMS for the specific use case.

Composing a DBMS and an SPE. The composing approach involves using both a DBMS and an SPE in conjunction while keeping them as separate components. This method offers flexibility, adaptability, and optimizability for various application requirements. The SPE primarily focuses on processing streams, while the DBMS handles state management, transactional support, and storage capabilities. Sharing state across queries and operators depends on the communication and synchronization between the SPE and DBMS. Durability can be achieved via the underlying DBMS, but the communication latency between the components can impact performance.

Examples of this approach include Storage Manager for Streams (SMS) [14]. Although composing a DBMS and an SPE is more complex than other approaches and introduces additional challenges, such as performance overhead and latency in communication between the systems, it provides a flexible and scalable solution for building TSP systems that can leverage the strengths of both components. The composing approach is well suited for applications requiring high levels of flexibility, adaptability, and performance, as well as the ability to separately optimize and customize the SPE and the DBMS components. It is particularly useful for applications with diverse and evolving requirements, as it enables the system to be easily adapted or extended to accommodate new functionality or optimizations. Examples of such applications include IoT systems, sensor networks, and other data-intensive applications with varying processing and storage requirements.

Advantages of this approach include: a) offering the most flexibility and adaptability by allowing separate optimization and customization of the SPE and the DBMS, enabling tailored solutions for specific application requirements; and b) providing better scalability and performance by distributing workloads across multiple systems and enabling parallelization of processing tasks. Disadvantages of this approach include: a) demanding additional effort to ensure integration correctness, which may increase development and maintenance complexity. This challenge can be mitigated by using standardized interfaces and middleware for communication between the SPE and the DBMS; and b) potentially exhibiting higher latency due to the need for communication and synchronization between the SPE and the DBMS. However, this can be mitigated by optimizing communication protocols and leveraging caching techniques to minimize data exchange overhead.

3.3.3 Performance metrics of TSP systems

In this sub-subsection, we discuss the key performance metrics and evaluation criteria for transactional stream processing systems. These metrics help in understanding the efficiency and effectiveness of various TSP architectures, as well as in identifying the trade-offs associated with different system designs.

Throughput. Throughput is a measure of the number of transactions or events processed per unit of time. High throughput is desirable in TSP systems, as it indicates the system's capability to handle large volumes of data efficiently. Throughput can be affected by factors such as system architecture, resource allocation, and workload characteristics. TSP systems should be designed to maximize throughput while maintaining other performance guarantees, such as low latency and correctness.

Latency. Latency is the time taken for a transaction or event to be processed by the TSP system, from the moment it enters the system until it is completely processed. Low latency is crucial for TSP systems, as many real-time applications require timely processing of data. Latency is influenced by factors such as system architecture, data processing complexity, and resource utilization. TSP systems should be designed to minimize latency, ensuring that transactions are processed quickly without compromising other performance aspects.

Scalability. Scalability measures the ability of a TSP system to handle increasing amounts of data and concurrent transactions without significant performance degradation. As the volume of data and the number of concurrent transactions grow, TSP systems must be able to scale to maintain low-latency access and fault tolerance. Techniques such as data

sharding, partitioning, and replication can be employed to distribute the shared mutable state across multiple nodes, providing horizontal scalability.

Correctness guarantee. Correctness guarantee in TSP systems refers to the extent to which a system can ensure data consistency and maintain the required transaction properties, as discussed in subsection 3.1, in the presence of failures, network delays, or other factors. This includes support for various isolation levels, consistency models, and durability guarantees. TSP systems should provide the appropriate level of correctness guarantees based on the specific requirements of the stream processing application.

- Isolation Levels: TSP systems should maintain proper isolation levels (e.g., serializable, snapshot isolation, read committed) to ensure that concurrent transactions do not interfere with each other and result in incorrect data processing.
- Consistency Models: Different TSP systems may adopt different consistency models (e.g., strong consistency, eventual consistency, or causal consistency) to provide a balance between data correctness and system performance. The choice of consistency model should be aligned with the application requirements and the tolerance for temporary inconsistencies.
- Durability Guarantees: TSP systems should ensure that once a transaction is committed, its effects are permanently recorded, even in the presence of failures. This can be achieved through techniques such as logging, checkpointing, and replication.

Balancing correctness guarantees with other performance metrics, such as throughput and latency, is crucial for achieving optimal performance in TSP systems. It is essential to recognize that the choice of the appropriate correctness guarantee is highly dependent on the application's requirements and the nature of the data being processed. For some applications, strong consistency and high isolation levels might be necessary, while for others, relaxed consistency models and lower isolation levels might be sufficient.

4 Systems offering transactional stream processing

We summarize six notable transactional stream processing (TSP) systems in Table 2. These TSP systems showcase the various approaches used to address the challenges in transactional stream processing. Each system offers unique features, and each has made some design choices that cater to different application requirements. Let us explore each of these systems in turn.

Table 2 Key characterization of six transactional stream processing systems

System	Unique features	Properties	Design aspects	Implementation details
STREAM	First unified framework	ACID, ordering, at-least-once	Unified transaction, lock-based approach	Embedding a DBMS in an SPE
Botan et al	Pipelined architecture	ACID, ordering	Unified transaction, per-tuple transactions	Embedding a DBMS in an SPE
S-Store	Static partitioning	ACID, ordering, exactly-once, at-least-once	State transaction, batch event and operator-based[a] triggered	Extending a DBMS
Braun et al	In-database stream analytics	ACID, ordering	State transaction, per-tuple transactions	Extending a DBMS
FlowDB	Transactional state management on Flink	ACID, ordering, exactly-once	State transaction, user-defined transaction boundary	Embedding a DBMS in an SPE
TStream/MorphStream	Dual-mode scheduling	ACID, ordering, exactly-once	State transaction, per-tuple transactions	Embedding a DBMS in an SPE

[a]Transactions are triggered by the stored procedure in each stream operator in S-Store

STREAM. STREAM [39] is an early TSP system that introduces a unified framework for continuous query processing over data streams and relations. It supports ACID properties and offers at-least-once delivery guarantees. STREAM's design approach is based on a combination of a sliding window model and a relational model for efficiently processing continuous queries. It employs a language called CQL (Continuous Query Language) to express queries, which can be translated into efficient query plans for execution. STREAM supports ACID properties and at-least-once delivery guarantees. The system focuses on ordering properties and state management properties by using windows and panes for efficient state handling. The architecture of STREAM is centered around a relational model, which simplifies the integration of stream processing and transaction processing.

Botan et al. Botan et al. [16] proposed a TSP system that focuses on a pipelined architecture for efficient parallel execution. The system supports ACID properties, but does not explicitly specify the delivery guarantees. Its design approach is centered around per-tuple transactions, which ensures strong consistency guarantees and high isolation. Botan et al. also address the challenges of implementing transactional guarantees in a stream processing environment, such as handling failures and maintaining consistency among shared mutable state. They propose techniques for addressing these challenges, including state replication, checkpointing, and recovery mechanisms. The system adheres to ACID properties, while delivery guarantees are not explicitly mentioned.

S-Store. S-Store [53] is a state transaction TSP system that builds upon H-Store, a distributed, main memory relational database system. It extends H-Store with stream processing capabilities and supports ACID properties. S-Store offers various delivery guarantees, such as exactly-once and at-least-once processing semantics. The system employs a partitioning mechanism for efficient parallel execution of stream processing tasks. In S-Store, each transaction is represented as a stored procedure, which can be invoked by incoming data streams. These stored procedures manipulate shared mutable states within the context of ACID-compliant transactions, ensuring consistency and isolation among concurrent tasks. S-Store supports ACID properties, exactly-once and at-least-once delivery guarantees, and employs a partitioning mechanism for distributed and parallelized execution. S-Store also focuses on fault tolerance techniques, including state replication, checkpointing, and recovery.

Braun et al. Braun et al. [17] presented a TSP system that focuses on in-database analytics by combining event processing and real-time analytics within the same database. The system supports ACID properties. However, the delivery guarantees are not explicitly specified. Its design approach is centered around in-database stream processing, which allows the system to efficiently handle complex event processing and real-time analytics tasks without requiring external tools or components. The system supports ACID properties, while delivery guarantees are not specified. The system also emphasizes performance metrics and evaluation of TSP systems, showcasing its efficiency in handling large volumes of data and complex analytics tasks.

FlowDB. FlowDB [3] is a TSP system that integrates stream processing and consistent state management. It supports ACID properties and provides exactly-once delivery guarantees. The system's design approach is based on a state transaction model, which allows for efficient management of shared mutable state. FlowDB features a language called FQL (Flow Query Language) for expressing continuous queries and supports the integration of custom stream processing operators and user-defined functions. FlowDB supports ACID properties, exactly-once delivery guarantees, and efficient state management. It also addresses fault tolerance and durability by employing checkpointing and recovery mechanisms.

TStream/MorphStream. TStream [88] and its successor MorphStream [50] are TSP systems that emphasize efficient concurrent state access on multi-core processors. These systems support ACID properties and offer exactly-once delivery guarantees. TStream's design approach includes a unique dual-mode scheduling strategy that combines transactional and parallel modes, to enable the system to maximize parallelism opportunities offered by modern multi-core architectures. TStream's dynamic restructuring execution strategy further improves concurrency by adapting the execution plan based on observed state access patterns during runtime.

5 Applications/scenarios leveraging TSP

TSP arises in varying domains, such as healthcare [84], the Internet of Things (IoT) [16], and e-commerce [3]). Table 3 summarizes thirteen scenarios. Each application encompasses diverse features, transactional models, and implementations of TSP systems. We can categorize them into four scenarios: *stream processing optimization*, *concurrent stateful processing*, *stream & DBMS integration*, and *recoverable stream processing*.

5.1 Stream processing optimization

Several works have proposed the consistent management of shared mutable states to optimize stream processing. Below we discuss these works in the context of stream processing optimization across four use cases: *sharing intermediate results*, *multi-query optimization*, *deterministic stream operations*, and *prioritizing query scheduling*.

Table 3 Summary of application scenarios and their demands of TSP properties

	Scenario	Ordering	ACID	State management	Reliability
Stream processing optimization					
1	Sharing Intermediate Results	Event, operation	ACID	Read–write, inter-operator	Consistency
2	Multi-Query Optimization	Event	ACID	Read–write, global	Delivery guarantee
3	Deterministic Stream Operations	Event	ACID	Read–write, intra- and Inter-operator	–
4	Prioritizing Query Scheduling	Event	ACID	Read–write, inter-query	Delivery guarantee
Concurrent stateful processing					
5	Ad hoc Queryable States	Event, operation	ACID	Read–write, global	Consistency, durability
6	Concurrent State Access	Event, operation	ACID	Read–write, intra- & Inter-operator	Consistency, durability
7	Active CEP	Event	ACID	Read–write, inter-query	Delivery guarantee, durability
Stream and DBMS integration					
8	Streaming Ingestion	Event	ACID	Read–write, global	Delivery guarantee, durability
9	Streaming OLTP	Operation	ACID	Read–write, per-transaction	Consistency, durability
10	Streaming OLAP	Event (optional)	Snapshot	Read–write, global	Delivery guarantee, fault tolerance
Robust stream processing					
11	Shared Persistent Storage	Operation	ACID	Read–write, global	Consistency, durability
12	Transaction Identifier	Operation	ACID	Read–write, global	Consistency, durability
13	Fault Tolerance Outsourcing	Operation	ACID	Read–write, global	Fault tolerance, durability

Sharing intermediate results. In the STREAM system [39], nearly identical states, or synopses, within a query plan are kept in a single store to reduce storage redundancy. Operators access their states exclusively via a stub interface. As operators are scheduled independently, they require slightly different data views, so STREAM employs a timestamp-based execution mechanism for correctness. Ordering properties are crucial in this scenario. Event ordering constraints maintain the data stream order, while operation ordering constraints ensure transaction operations' order. ACID properties maintain the correctness and consistency of shared mutable states. State management properties require read–write state types and inter-operator access scope for managing shared states among different operators.

Multi-query optimization. Ray et al. [64] introduced the SPASS (Scalable Pattern Sharing on Event Streams) framework, which optimizes time-based event correlations among queries and shares processing effectively. The optimizer identifies a shared pattern plan, maintaining an optimality bound. The runtime then uses shared continuous sliding view technology for executing the shared pattern plan. A sequence transaction model on shared views defines the correctness of concurrent shared pattern execution. SPASS does not modify existing states, but selects, inserts, and deletes shared states like sliding views. Ordering properties are crucial for maintaining the correct order of pattern queries. Event ordering constraints preserve data stream order, while operation ordering constraints are less important as sharing is among queries. ACID properties maintain consistency and correctness of shared states like sliding views. State management properties need read–write state types and global access scope for managing shared states among pattern queries. However, some implementation details are not specified in the original paper (e.g., sliding views' data layout, key used for searching shared sliding views).

Deterministic stream operations. Handling out-of-order streams is often a performance bottleneck due to the conflict between data parallelism and order-sensitive processing. While data parallelism improves throughput by processing events concurrently, it can cause events to be handled out of order. Most solutions use locks or non-lock algorithms like sorting [89]. Brito et al. [18] proposed an interesting non-lock approach using software transactional memory (STM) for stream processing. They model processing a batch of input data at order-sensitive operators as a transaction and pre-assign commit timestamps, effectively imposing order. Events received out of order or conflicting are processed in parallel optimistically, but are not output until all preceding events are completed, ensuring consistent operator states. Ordering properties are crucial for managing out-of-order streams, with event ordering constraints needed to maintain data stream order. Operation ordering constraints are less relevant. ACID properties maintain the correctness and consistency of shared mutable states, especially when handling out-of-order and conflicting events. State management properties require read–write state types and intra- and inter-operator access scope for managing shared states within and among different operators.

Prioritizing query scheduling. Handling potentially infinite data streams requires continuous queries with window constraints to limit tuple processing. Most implementations execute sliding window queries and window updates serially, implicitly assuming a window cannot be advanced while accessed by a query. Golab et al. [36] argue that concurrent processing of queries (reads) and window updates (writes) is necessary for prioritized query scheduling to improve answer freshness. They model window updates and queries as transactions with atomic subwindow reads and writes, which can lead to read–write conflicts. Golab et al. [36] prove traditional conflict serializability is insufficient and define stronger isolation levels restricting allowed serialization orders following event ordering. Ordering properties are crucial for prioritizing query scheduling, with event ordering constraints ensuring correct data stream order. ACID properties maintain correctness and consistency of shared mutable states when handling concurrent reads and writes. State management properties require read–write state types and inter-query access scope for managing shared states among different queries and window updates.

5.2 Concurrent stateful processing

In this scenario, application workloads consist of both ad hoc and continuous queries, which may access and modify common application states for future reference [19]. We will discuss three representative applications in the context of concurrent stateful processing: *ad hoc queryable states*, *concurrent state access*, and *active complex event processing*, focusing on the properties of TSP required by each application.

Ad hoc queryable states. Ad hoc queries, or snapshot queries, can be submitted to an SPE anytime, executed once, and provided insights into the system's current state. They may be used to obtain further details in response to continuous query result changes. In Botan et al.'s example [16], real-time sensors generate temperature measurements to ensure temperature-sensitive devices operate within design specifications. When a temperature reading falls out of the operating range, it triggers an alert. The SPE must ensure table updates and stream temperature readings are executed in the correct order, demanding event and operation ordering constraints. To maintain the specifications table's integrity and prevent data corruption, ACID properties are necessary. The state is read–write, as the table needs updates, and the access scope is global, as all incoming temperature readings need to probe the table. Delivery guarantees and fault tol-

erance are vital for ensuring accurate temperature reading processing and system recovery from potential failures.

Concurrent state access. In applications like Ververica Streaming Ledger [33] (SL), operators like parser, deposit, transfer, and sink may need to share access to states, such as account and asset data. To process transactions correctly and maintain data consistency, event and operation ordering constraints are crucial. ACID properties are necessary to ensure shared state accuracy and prevent data corruption during concurrent access. The state is read–write, as the account and asset data need updates, and the access scope includes intra- and inter-operator, as multiple operators and their replica instances access and modify the shared stat. Delivery guarantees, fault tolerance, and durability are critical for maintaining accurate and consistent transaction processing. CAP Theorem considerations must be taken into account when designing the system to balance consistency, availability, and partition tolerance.

Active complex event processing. In the realm of stream processing, active complex event processing is a method that continuously monitors and analyzes a series of real-time events to detect certain patterns or sequences. Wang et al. [84] have applied this concept to a hospital infection control application. The system uses sensor devices to generate real-time data on healthcare workers' (HCWs) behaviors such as "exit," "sanitize," and "enter." These events are then analyzed by pattern queries that aim to detect any violations of hospital hygiene rules. The status of all HCWs, whether static or dynamic, is stored in tables. In this application, it is crucial to maintain the correct order of events and handle concurrent accesses and updates during stream execution. This requires event ordering constraints and ACID properties to ensure data consistency and prevent conflicts. The state management in this scenario involves read–write states with inter-query access scope, as multiple pattern queries may read or update the tables concurrently. Reliability and delivery guarantees are essential for accurately and consistently detecting violations of hospital hygiene rules.

5.3 Stream and DBMS integration

The integration of SPEs with DBMSs is becoming increasingly important [75]. Scenarios such as stream data ingestion (i.e., *Streaming Ingestion*), implementing OLTP queries in alternative ways (*Streaming OLTP*), and mixed stream and analytic queries (*Streaming OLAP*) can be well supported by TSP systems. In the following sections, we discuss each of these scenarios and their requirements concerning TSP properties.

Streaming ingestion. Streaming ingestion is an essential process for organizations handling large volumes of data, as it enables more timely access to incremental results compared to traditional batch ingestion. This approach processes smaller micro-batches throughout the day, which requires proper management of ordering and state properties. A notable example by Meehan et al. [52] involves self-driving vehicles, where the value of sensor data decreases over time. In this case, timely processing and storage of time series data are critical for making valuable decisions. The authors adapt TPC-DI [63], a standard benchmark for data ingestion, to assess streaming ingestion effectiveness while considering new data dependencies introduced by breaking large batches into smaller ones. In this scenario, maintaining the correct order of time series data is crucial, necessitating event ordering constraints. ACID properties are required to ensure the correctness and consistency of ingested data, especially with high sample rates and large data volumes. Read–write state types are needed for efficient management and persistence of ingested data. Delivery guarantees are essential for processing and persisting ingested data accurately and consistently in near real time. Additionally, durability is necessary to ensure data availability for analysis and decision-making, even after a system failure or outage.

Streaming OLTP. Streaming OLTP addresses traditional OLTP workloads using streaming queries. Chen and Migliavacca [26] propose *StreamDB*, a TSP-based system. Streaming OLTP requires ACID properties, state management properties (e.g., database partitioning), and fault tolerance. *StreamDB* uses three operators in a streaming query: (1) *Source* operator, receiving transactions and sending them to downstream data operators; (2) *Data* operator, managing a portion of a database, executing transactions, and producing results; and (3) *Sink* operator, receiving transaction responses. *StreamDB* reduces lock contention by distributing the database among multiple data operators. However, creating an optimal stream dataflow graph for diverse OLTP workloads remains an open question. In this scenario, operation ordering constraint is crucial for correct transaction processing and minimizing lock contention. ACID properties ensure database correctness and consistency during transaction processing. Read–write state types are required for managing the database and transactions, while maintaining consistency. Consistency, durability, and fault tolerance are vital for accurate and consistent transaction processing and for maintaining database consistency even after system failures or outages.

Streaming OLAP. Organizations often need real-time analysis of data streams for immediate decision-making, and several related applications have been described in the literature [17, 38, 70]. The Huawei-AIM workload [17] features a three-tier architecture with storage, an SPE, and real-time analytics (RTA) nodes. RTA nodes push analytical queries to storage nodes, merge partial results, and deliver final results to clients. To meet the service level objective (SLO), a consistent state (or snapshot) must not be older than a certain bound. In this scenario, event ordering constraints ensure

correct analytical query processing. ACID properties can be relaxed (e.g., snapshot isolation) to maintain data correctness and consistency. Read–write state types manage the database and ensure the consistent state meets the SLO. Delivery guarantees, consistency, durability, and fault tolerance ensure accurate query processing, data persistence, and system recovery from failures.

5.4 Robust stream processing

In addition to performance requirements such as *scalability* and *low latency*, many critical streaming applications demand SPEs to recover quickly from failures [73]. Consequently, considerable effort has been dedicated to achieving fault tolerance in SPEs. We discuss previous attempts to employ transaction-like concepts to ensure high availability and fault tolerance in stream processing, focusing on their demands for properties of TSP discussed in Sect. 3.1.

Shared persistent storage. MillWheel [6] uses an event-driven API for stateful computations and stores input and output data persistently. It relies on remote storage systems like BigTable [24] for managing state updates and handling fault tolerance through data replication. This relates to durability in ACID properties and reliability and delivery guarantees. MillWheel ensures atomicity by encapsulating all per-key updates in a single atomic operation. While it enforces operation ordering constraints, strict event ordering constraints are not provided. State management properties are vital in MillWheel due to its dependence on shared persistent storage.

Transaction identifier. Trident's "transactional topology" [51] processes small batches of tuples as a single operation and assigns unique transaction identifiers (*TXID*), relating to ordering properties. TXID, logged in external storage along with operator state, addresses atomicity and durability in ACID properties. If TXID mismatch occurs, a batch must be resubmitted, requiring strict transaction processing ordering (operation ordering constraint) and potentially limiting throughput. State management properties are essential in this approach, as it depends on read–write states. Trident provides strong delivery guarantees and fault tolerance, but may lose intermediate results during failures due to disregarding buffered input states.

Fault tolerance outsourcing. Ishikawa et al. [41] propose integrating fault tolerance into an OLTP engine for data stream processing. This involves backing up data streams in an in-memory database system instead of a file system, addressing durability in ACID properties and reliability and delivery guarantees. It enforces operation ordering constraints, similar to H-Store's state partitioning transaction processing. However, outsourcing fault tolerance can strain the remote store during data spikes, potentially impacting other applications sharing the store. This approach also requires CAP theorem considerations, as the choice of an in-memory database system might involve trade-offs between consistency and availability.

Remark 5 (TSP is not just nice to have, but sometimes a must) By employing modern SPEs, existing workarounds, such as using external databases to store shared application states, can lead to significant additional programming effort [84], poor performance [53], and even incorrect results [22, 36]. This problem is exacerbated if more complex shared mutable state storage and retrieval queries such as range lookups are further required. In contrast, TSP systematically manages concurrent accesses to shared application states with transactional correctness. This even leads TSP to have the potential of better support for traditional database workloads (i.e., OLTP and OLAP) as well.

Remark 6 (TSP-based applications have diverse requirements) Some TSP-based applications do not require insertion and deletion operations at all, while others do not need to update shared mutable states. Also noteworthy is that transactional dependency is rare among applications, which means that in most cases, the input parameters in a transaction are predetermined from the triggering events. Targeting a narrowed application domain, a TSP system can take advantage of these diverse requirements to simplify its design and improve system performance. To date, there is no standard benchmark for TSP systems [76], which must include comprehensive performance metrics, diverse workload features, and meaningful application scenarios. The applications that we list in Table 3 may serve as a starting point for the construction of a standard benchmark. However, more applications may need to be included, arranged according to their particular application's features.

6 Research outlook

In this section, we offer a perspective on future research directions of TSP.

6.1 Novel applications

The rise of IoT generates real-time data that needs immediate processing. Traditional big data applications were designed for large static datasets, but modern applications demand more. We foresee novel streaming applications benefiting from TSP solutions as the range of applications served by SPEs widens. Current research, like NebulaStream [87], explores systems meeting these requirements. We discuss various application areas, including online machine learning/stream mining, mixed batch/stream transactional workloads, streaming materialized views, and cloud applications.

Optimization for stateful stream processing. Shahvarani and Jacobsen's IBWJ [68] accelerates sliding window joins by using a shared index data structure, reducing redundant memory access and improving performance. As new tuples arrive, the index structure is updated, raising concurrency control issues. Shahvarani et al. [68] propose a low-cost concurrency control mechanism for high-rate update queries. A TSP system could naturally handle this concurrency problem, providing durability when required. Despite its potential, we are unaware of any practical implementation of this approach.

Online machine learning/stream mining. The rising demand for data stream analysis necessitates online learning and mining. Current efforts support continuous queries (CQs) referencing non-streaming resources like databases and ML models [59]. Model-based streaming systems, such as anomaly detectors, require regular model updates without significantly increasing operational costs [6]. Due to the lack of transactional support in traditional SPEs, implementing emerging streaming learning and mining algorithms can be challenging [66]. Although existing batch-based ML training (like TensorFlow) may not need to care for inconsistencies in the state they handle, a streaming ML scenario may prohibit such inconsistencies as each input data may be allowed to be used, up to a limited threshold, and any inconsistency may lead to significantly lower training quality. It thus remains an interesting future work to study how those novel training and mining use cases can be supported efficiently in TSP systems, which bring features, such as elastic scaling, fault tolerance guarantees, and shared state consistency to users, even at the virtual space [56].

Mixed batch/stream transactional workloads. Many enterprise applications, particularly in finance and IoT, generate mixed workloads with continuous stream processing, OLTP, and OLAP. DeltaLake [8] allows streaming jobs to write small objects into a table with low latency and coalesce them into larger objects later. Fast "tailing" reads are also supported for treating a Delta table as a message bus. Tatbul [75] outlines challenges in streaming data integration, including common semantic models, optimization, and transactional issues. These challenges persist due to diverse applications and systems focusing on limited feature sets.

Streaming materialized views. Traditional materialized views (MVs) are not optimized for high-velocity data stream processing, leading to the need for streaming materialized views (SMVs). SMVs must handle high-velocity inputs, update states with random access patterns, and share updated states among concurrent entities. Recent works, such as S-Query [79] and Umbra's continuous view scheme [85], have proposed solutions, but the former lacks strict ACID guarantees, and the latter has yet to be compared with state-of-the-art transactional stream processing systems like S-Store [53], TStream [88], and TSpoon [4]. The distinction in use cases drives a clear separation of concerns and further investigation into optimizing SMVs.

Cloud applications. As Carbone et al. observed [20], existing SPEs frequently lack transactional features that are essential for Cloud applications, which require advanced business logic and coordination. This is true in particular when the state schema changes frequently, necessitating reliable versioning of state to maintain consistency. A specific use case, known as *stateful function-as-a-service* (FaaS) [5, 42], exemplifies these requirements, including ACID transactions, global state consolidation, and the capacity for debugging and auditing. Surprisingly, these demands align closely with the needs of transactional stream processing (TSP), especially concerning transactional shared state management during stream processing. However, it is important to clarify that current stateful FaaS solutions typically defer transactional guarantees to the application layer, a gap that underscores the relevance of TSP approaches. Yet, it remains an open question whether existing TSP systems like S-Store [53] can fully meet the nuanced demands of Cloud applications. Questions such as how to support debugging [44] and isolation [71] in stateful stream processing, especially when structured as micro-services, present intriguing challenges worthy of further investigation.

6.2 Novel hardware platforms

Modern hardware advancements have made servers with hundreds of cores and several terabytes of main memory available. Such advancements have driven researchers to rethink TSP systems and put emerging hardware platforms to good use [89]. Next, we take a closer look at multi-/many-core architectures, nonvolatile storage, and trusted computing platforms.

Multi-/many-core architectures. Supporting shared mutable states in TSP systems can create bottlenecks due to concurrent state accesses. TStream [88] is a recent example that effectively utilizes multi-core CPUs to improve concurrent shared state access performance through dual-mode scheduling and a dynamic transaction restructuring mechanism. However, current TSP systems still face scalability challenges with complex workloads and input dependencies. Further research is needed to enhance TSP systems for complex workloads, emerging multi-/many-core architectures with high-bandwidth memory, and multi-node settings while maintaining correctness guarantees [76].

Nonvolatile storage. Nonvolatile memory (NVM) is an emerging technology offering byte-addressability and low latency of DRAM along with persistence and density of block-based storage media, but with limited cell endurance and read–write latency asymmetry. Fernando et al. [62] explored efficient approaches for analytical workloads on NVM, potentially laying the foundation for future TSP sys-

tems [67]. NVMe-based SSDs can deliver high performance in terms of latency and peak bandwidth. Lee et al. [46] investigated performance limitations of SPEs managing application states on SSDs, showing query-aware optimization can improve stateful stream processing on SSDs. Their work is valuable for TSP systems with strict ACID and streaming property requirements, but more research is needed.

Trusted computing platforms. The need for low latency and local processing of sensitive IoT data calls for edge stream processing. However, edge devices are vulnerable to attacks due to limited power and computing capacity, posing severe security threats to sensitive data. A potential solution [60] is trusted computing platforms (TCPs), which protect data and code within isolated, encrypted memory areas. Bringing TSP to TCPs is non-trivial and requires further research, particularly in handling limited memory for transactional stateful stream processing [54]. Additionally, scaling systems to multiple TCPs in a distributed environment presents challenges due to each computing node's computational limits.

7 Conclusion

In this survey, we provided a comprehensive overview of transactional stream processing (TSP), and addressed key concepts, techniques, and challenges to be overcome, in order to ensure reliable and consistent data stream processing. We introduced terms, definitions, and a conceptual framework for TSP systems and presented a taxonomy that offers a structured understanding of various approaches and models for integrating transactional properties with streaming requirements. We also discussed several notable TSP systems, each showcasing unique features and design choices that were made to cater to different application requirements. These systems offer insight and inform designers about the alternative choices they will need to make when designing and implementing a novel TSP system. We highlighted various TSP applications and use cases, such as stream processing optimization, concurrent stateful processing, and stream and DBMS integration. Finally, we explored some open challenges and suggest future directions for TSP research and development, including novel applications and hardware platforms. This survey serves as a resource for researchers and practitioners. It aims to inspire others to pursue work in this field and develop efficient, reliable, and scalable TSP systems for diverse application domains.

Acknowledgements This work is supported by the National Research Foundation, Singapore, and the Infocomm Media Development Authority under its Future Communications Research & Development Programme (FCP-SUTD-RG-2021-005), the SUTD Start-up Research Grant (SRT3IS21164), the DFG Priority Program (MA4662-5), the German Federal Ministry of Education and Research (BMBF) under grants 01IS18025A (BBDC - Berlin Big Data Center) and 01IS18037A (BIFOLD - Berlin Institute for the Foundations of Learning and Data). Shuhao Zhang' work is partially done while working as a Postdoc at TU Berlin.

Declarations

Declarations The authors have no financial or proprietary interests in any material discussed in this article.

References

1. Abadi, D.J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., et al.: The design of the borealis stream processing engine. CIDR'05 **5**, 277–289 (2005)
2. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: a new model and architecture for data stream management. VLDB J. **12**(2), 120–139 (2003). https://doi.org/10.1007/s00778-003-0095-z
3. Affetti, L., Margara, A., Cugola, G.: Flowdb: Integrating stream processing and consistent state management. In: Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, Debs '17, pp. 134–145. Acm, New York, NY, USA (2017). https://doi.org/10.1145/3093742.3093929
4. Affetti, L., Margara, A., Cugola, G.: Tspoon: Transactions on a stream processor. J. Parallel Distrib. Comput. **140**, 65–79 (2020) https://doi.org/10.1016/j.jpdc.2020.03.003. www.sciencedirect.com/science/article/pii/S0743731518305082
5. Akhter, A., Fragkoulis, M., Katsifodimos, A.: Stateful functions as a service in action. Proc. VLDB Endow. **12**(12), 1890–1893 (2019)
6. Akidau, T., Balikov, A., Bekiroğlu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., Whittle, S.: Millwheel: Fault-tolerant stream processing at internet scale. Proc. VLDB Endow. **6**(11), 1033–1044 (2013). https://doi.org/10.14778/2536222.2536229
7. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. VLDB J. **15**(2), 121–142 (2006). https://doi.org/10.1007/s00778-004-0147-z
8. Armbrust, M., Das, T., Sun, L., Yavuz, B., Zhu, S., Murthy, M., Torres, J., van Hovell, H., Ionescu, A., Łuszczak, A., Świtakowski, M., Szafrański, M., Li, X., Ueshin, T., Mokhtar, M., Boncz, P., Ghodsi, A., Paranjpye, S., Senster, P., Xin, R., Zaharia, M.: Delta lake: high-performance acid table storage over cloud object stores. Proc. VLDB Endow. **13**(12), 3411–3424 (2020). https://doi.org/10.14778/3415478.3415560
9. Ayad, A.M., Naughton, J.F.: Static optimization of conjunctive queries with sliding windows over infinite streams. In: Proceedings

of the 2004 ACM SIGMOD International Conference on Management of Data, SIGMOD '04, pp. 419–430. Association for Computing Machinery, New York, NY, USA (2004). https://doi.org/10.1145/1007568.1007616
10. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '02, pp. 1–16. Association for Computing Machinery, New York, NY, USA (2002). https://doi.org/10.1145/543613.543615
11. Balazinska, M., Balakrishnan, H., Madden, S., Stonebraker, M.: Fault-tolerance in the borealis distributed stream processing system. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, pp. 13–24 (2005)
12. Bernstein, P., Newcomer, E.: Principles of Transaction Processing: For the Systems Professional. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1997)
13. Bernstein, P.A., Goodman, N.: Concurrency control in distributed database systems. ACM Comput. Surv. 1981 **13**(2), 185–221 (1981). https://doi.org/10.1145/356842.356846
14. Botan, I., Alonso, G., Fischer, P.M., Kossmann, D., Tatbul, N.: Flexible and scalable storage management for data-intensive stream processing. In: Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, EDBT '09, pp. 934–945. Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1516360.1516467
15. Botan, I., Cho, Y., Derakhshan, R., Dindar, N., Haas, L., Kim, K., Lee, C., Mundada, G., Shan, M.C., Tatbul, N., Yan, Y., Yun, B., Zhang, J.: Design and implementation of the maxstream federated stream processing architecture. Tech. Rep. ETH Zurich Dep. Comput. Sci. (2009). https://doi.org/10.1007/978-3-642-14559-9_2
16. Botan, I., Fischer, P.M., Kossmann, D., Tatbul, N.: Transactional stream processing. In: Proceedings of the 15th International Conference on Extending Database Technology, EDBT '12, pp. 204–215. Acm, New York, NY, USA (2012). https://doi.org/10.1145/2247596.2247622
17. Braun, L., Etter, T., Gasparis, G., Kaufmann, M., Kossmann, D., Widmer, D., Avitzur, A., Iliopoulos, A., Levy, E., Liang, N.: Analytics in motion: High performance event-processing and real-time analytics in the same database. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15, pp. 251–264. Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2723372.2742783
18. Brito, A., Fetzer, C., Sturzrehm, H., Felber, P.: Speculative out-of-order event processing with software transaction memory. In: R. Baldoni (ed.) Proceedings of the Second International Conference on Distributed Event-Based Systems, DEBS 2008. Rome, Italy, July 1–4, 2008, ACM International Conference Proceeding Series **332**, 265–275 (2008). https://doi.org/10.1145/1385989.1386023. (**ACM**)
19. Carbone, P., Ewen, S., Fóra, G., Haridi, S., Richter, S., Tzoumas, K.: State management in apache flink: Consistent stateful distributed stream processing. Proc. VLDB Endow. **10**(12), 1718–1729 (2017). https://doi.org/10.14778/3137765.3137777
20. Carbone, P., Fragkoulis, M., Kalavri, V., Katsifodimos, A.: Beyond analytics: The evolution of stream processing systems. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20, pp. 2651–2658. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3318464.3383131
21. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink: Stream and batch processing in a single engine. Bull. IEEE Comput. Soc. Techn. Comm. Data Eng. **36**(4), 1–12 (2015)
22. Cetintemel, U., Du, J., Kraska, T., Madden, S., Maier, D., Meehan, J., Pavlo, A., Stonebraker, M., Sutherland, E., Tatbul, N., Tufte, K., Wang, H., Zdonik, S.: S-store: A streaming newsql system for big velocity applications. Proc. VLDB Endow. **7**(13), 1633–1636 (2014). https://doi.org/10.14778/2733004.2733048
23. Chandrasekaran, S., Franklin, M.: Remembrance of streams past: Overload-sensitive management of archived streams. In: VLDB (2004)
24. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. In: 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 205–218 (2006)
25. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. (TOCS) **26**(2), 1–26 (2008)
26. Chen, H., Migliavacca, M.: Streamdb: A unified data management system for service-based cloud application. In: 2018 IEEE International Conference on Services Computing (SCC), pp. 169–176. IEEE (2018)
27. Chen, Q., Hsu, M.: Experience in extending query engine for continuous analytics. In: T. Bach Pedersen, M.K. Mohania, A.M. Tjoa (eds.) Data Warehousing and Knowledge Discovery, pp. 190–202. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
28. Chen, Q., Hsu, M.: Query engine grid for executing sql streaming process. In: International Conference on Data Management in Grid and P2P Systems, pp. 95–107. Springer (2011)
29. Chen, Q., Hsu, M., Zeller, H.: Experience in continuous analytics as a service (caaas). In: Proceedings of the 14th International Conference on Extending Database Technology, EDBT/ICDT '11, pp. 509–514. Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1951365.1951426
30. Conway, N.: Cisc 499*: Transactions and data stream processing. Apr **6**, 28 (2008)
31. Coral8, inc, http://www.coral8.com/ (2008)
32. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J.J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al.: Spanner: Google's globally distributed database. ACM Trans. Comput. Syst. (TOCS) **31**(3), 1–22 (2013)
33. Data Artisans Streaming Ledger Serializable ACID Transactions on Streaming Data, https://www.data-artisans.com/blog/serializable-acid-transactions-on-streaming-data (2018)
34. Franklin, M., Krishnamurthy, S., Conway, N., Li, A., Russakovsky, A., Thombre, N.: Continuous analytics: Rethinking query processing in a network-effect world. In: CIDR (2009)
35. Garcia-Molina, H., Salem, K.: Sagas. SIGMOD Rec. **16**(3), 249–259 (1987). https://doi.org/10.1145/38714.38742
36. Golab, L., Bijay, K.G., Özsu, M.T.: On concurrency control in sliding window queries over data streams. In: Ioannidis, Y., Scholl, M.H., Schmidt, J.W., Matthes, F., Hatzopoulos, M., Boehm, K., Kemper, A., Grust, T., Boehm, C. (eds.) Advances in Database Technology - EDBT 2006, pp. 608–626. Springer Berlin Heidelberg, , Berlin, Heidelberg (2006)
37. Golab, L., Özsu, M.T.: Update-pattern-aware modeling and processing of continuous queries. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, SIGMOD '05, p. 658-669. Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1066157.1066232
38. Götze, P., Sattler, K.: Snapshot isolation for transactional stream processing. In: EDBT (2019)
39. S., Group, et al.: Stream: the Stanford stream data manager. Tech. Rep., Stanford InfoLab (2003)
40. Gürgen, L., Roncancio, C., Labbé, C., Olive, V.: Transactional issues in sensor data management. In: Proceedings of the 3rd Workshop on Data Management for Sensor Networks: In Conjunction

with VLDB 2006, DMSN '06, pp. 27–32. Association for Computing Machinery, New York, NY, USA (2006). https://doi.org/10.1145/1315903.1315910
41. Ishikawa, Y., Sugiura, K., Takao, D.: Fault tolerant data stream processing in cooperation with OLTP engine. In: Mondal, A., Gupta, H., Srivastava, J., Reddy, P.K., Somayajulu, D. (eds.) Big Data Anal., pp. 3–14. Springer International Publishing, Cham (2018)
42. Katsifodimos, A., Fragkoulis, M.: Operational stream processing: Towards scalable and consistent event-driven applications. (2019)
43. Krishnamurthy, S., Franklin, M.J., Davis, J., Farina, D., Golovko, P., Li, A., Thombre, N.: Continuous analytics over discontinuous streams. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, pp. 1081–1092. Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1807167.1807290
44. Kumar, A., Wang, Z., Ni, S., Li, C.: Amber: a debuggable dataflow system based on the actor model. Proc. VLDB Endow. **13**(5), 740–753 (2020). https://doi.org/10.14778/3377369.3377381
45. Kumar, D., Li, J., Chandra, A., Sitaraman, R.: A TTL-based approach for data aggregation in geo-distributed streaming analytics. Proc. ACM Measure. Anal. Comput. Syst. **3**(2), 1–27 (2019)
46. Lee, G., Eo, J., Seo, J., Um, T., Chun, B.G.: High-performance stateful stream processing on solid-state drives. In: Proceedings of the 9th Asia-Pacific Workshop on Systems, APSys '18, pp. 9:1–9:7. ACM, New York, NY, USA (2018). https://doi.org/10.1145/3265723.3265739
47. Liarou, E., Goncalves, R., Idreos, S.: Exploiting the power of relational databases for efficient stream processing. In: Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, pp. 323–334 (2009)
48. Liarou, E., Kersten, M.: Datacell: Building a data stream engine on top of a relational database kernel. In: VLDB PhD Workshop (2009)
49. Madden, S., Franklin, M.J.: Fjording the stream: An architecture for queries over streaming sensor data. In: Proceedings 18th International Conference on Data Engineering, pp. 555–566. IEEE (2002)
50. Mao, Y., Zhao, J., Zhang, S., Liu, H., Markl, V.: Morphstream: Adaptive scheduling for scalable transactional stream processing on multicores. In: Proceedings of the 2023 International Conference on Management of Data (SIGMOD), SIGMOD '23. Association for Computing Machinery, New York, NY, USA (2023)
51. Marz, N.: Trident API Overview: https://github.com/nathanmarz/storm/wiki/
52. Meehan, J., Aslantas, C., Zdonik, S., Tatbul, N., Du, J.: Data ingestion for the connected world. In: CIDR (2017)
53. Meehan, J., Tatbul, N., Zdonik, S., Aslantas, C., Cetintemel, U., Du, J., Kraska, T., Madden, S., Maier, D., Pavlo, A., Stonebraker, M., Tufte, K., Wang, H.: S-store: streaming meets transaction processing. Proc. VLDB Endow. **8**(13), 2134–2145 (2015). https://doi.org/10.14778/2831360.2831367
54. Meftah, S., Zhang, S., Veeravalli, B., Aung, K.M.M.: Revisiting the design of parallel stream joins on trusted execution environments. Algorithms **15**(6), 183 (2022). https://doi.org/10.3390/a15060183
55. Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., Varma, R.: Query processing, resource management, and approximation ina data stream management system. In: CIDR 2003. Stanford InfoLab (2002)
56. Ooi, B.C., Tan, K.L., Tung, A., Chen, G., Shou, M.Z., Xiao, X., Zhang, M.: Sense the physical, walkthrough the virtual, manage the metaverse: A data-centric perspective. arXiv preprint arXiv:2206.10326 (2022)
57. Oyamada, M., Kawashima, H., Kitagawa, H.: Efficient invocation of transaction sequences triggered by data streams. In: 2011 International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, pp. 332–337. IEEE (2011)
58. Oyamada, M., Kawashima, H., Kitagawa, H.: Continuous query processing with concurrency control: Reading updatable resources consistently. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, pp. 788–794. Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2480362.2480514
59. Oyamada, M., Kawashima, H., Kitagawa, H.: Data stream processing with concurrency control. SIGAPP Appl. Comput. Rev. **13**(2), 54–65 (2013). https://doi.org/10.1145/2505420.2505425
60. Park, H., Zhai, S., Lu, L., Lin, F.X.: Streambox-tz: Secure stream analytics at the edge with trustzone. In: Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '19, pp. 537–554. USENIX Association, USA (2019)
61. Pekhimenko, G., Guo, C., Jeon, M., Huang, P., Zhou, L.: {TerseCades}: Efficient data compression in stream processing. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18), Usenix Atc '18, pp. 307–320. USENIX Association, Berkeley, CA, USA (2018). http://dl.acm.org/citation.cfm?id=3277355.3277385
62. Philipp, G., Stephan, B., Kai-Uwe, S.: An nvm-aware storage layout for analytical workloads. In: 2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW), pp. 110–115 (2018). https://doi.org/10.1109/icdew.2018.00025
63. Poess, M., Rabl, T., Jacobsen, H.A., Caufield, B.: TPC-DI: the first industry benchmark for data integration. Proc. VLDB Endow. **7**(13), 1367–1378 (2014). https://doi.org/10.14778/2733004.2733009
64. Ray, M., Lei, C., Rundensteiner, E.A.: Scalable pattern sharing on event streams*. In: Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, pp. 495–510. Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2882903.2882947
65. Rocksdb. http://rocksdb.org/
66. Sahin, O.C., Karagoz, P., Tatbul, N.: Streaming event detection in microblogs: Balancing accuracy and performance. In: Bakaev, M., Frasincar, F., Ko, I.Y. (eds.) Web Eng., pp. 123–138. Springer International Publishing, Cham (2019)
67. Sattler, K.U.: Transactional stream processing on non-volatile memory (2019). https://www.tu-ilmenau.de/dbis/research/active-projects/transactional-stream-processing/
68. Shahvarani, A., Jacobsen, H.A.: Parallel index-based stream join on a multicore cpu. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20, pp. 2523–2537. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3318464.3380576
69. Shaikh, S.A., Chao, D., Nishimura, K., Kitagawa, H.: Incremental continuous query processing over streams and relations with isolation guarantees. In: Proceedings, Part I, 27th International Conference on Database and Expert Systems Applications - Vol. 9827, DEXA 2016, pp. 321–335. Springer-Verlag, Berlin, Heidelberg (2016). https://doi.org/10.1007/978-3-319-44403-1_20. https://doi.org/10.1007/978-3-319-44403-1%5F20
70. Shaikh, S.A., Kitagawa, H.: Streamingcube: seamless integration of stream processing and olap analysis. IEEE Access **8**(104), 104632–104649 (2020). https://doi.org/10.1109/ACCESS.2020.2999572
71. Shillaker, S., Pietzuch, P.: Faasm: Lightweight isolation for efficient stateful serverless computing. In: 2020 USENIX Annual Technical Conference (USENIX ATC 20), pp. 419–433. USENIX Association (2020). https://www.usenix.org/conference/atc20/presentation/shillaker
72. Silvestre, P.F., Fragkoulis, M., Spinellis, D., Katsifodimos, A.: Clonos: Consistent Causal Recovery for Highly-Available Streaming Dataflows, p. 1637-1650. Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3448016.3457320

73. Stonebraker, M., Çetintemel, U., Zdonik, S.: The 8 requirements of real-time stream processing. SIGMOD Rec. **34**(4), 42–47 (2005). https://doi.org/10.1145/1107499.1107504
74. Stonebraker, M., Madden, S., Abadi, D.J., Harizopoulos, S., Hachem, N., Helland, P.: The end of an architectural era: (it's time for a complete rewrite). In: Proc VLDB Endow. 2007
75. Tatbul, N.: Streaming data integration: Challenges and opportunities. In: 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010), pp. 155–158 (2010). https://doi.org/10.1109/ICDEW.2010.5452751
76. Tatbul, N.: Transactional Stream Processing, pp. 4205–4211. Springer New York, New York, NY (2018). https://doi.org/10.1007/978-1-4614-8265-9_80704. https://doi.org/10.1007/978-1-4614-8265-9%5F80704
77. Theodorakis, G., Kounelis, F., Pietzuch, P., Pirk, H.: Scabbard: Single-node fault-tolerant stream processing. Proc. VLDB Endow. **15**(2), 361–374 (2021). https://doi.org/10.14778/3489496.3489515
78. To, Q.C., Soto, J., Markl, V.: A survey of state management in big data processing systems. VLDB J. **27**(6), 847–872 (2018). https://doi.org/10.1007/s00778-018-0514-9
79. Verheijde, J., Karakoidas, V., Fragkoulis, M., Katsifodimos, A.: S-query: Opening the black box of internal stream processor state. In: 2022 IEEE 38th International Conference on Data Engineering (ICDE), pp. 1314–1327. IEEE (2022)
80. Vidyasankar, K.: Transactional properties of compositions of internet of things services. pp. 1–6 (2015). https://doi.org/10.1109/ISC2.2015.7366218
81. Vidyasankar, K.: A transaction model for executions of compositions of internet of things services. Proc. Comput. Sci. **83**, 195–202 (2016). https://doi.org/10.1016/j.procs.2016.04.116
82. Vidyasankar, K.: Transactional composition of executions in stream processing. In: 2016 27th International Workshop on Database and Expert Systems Applications (DEXA), pp. 114–118 (2016). https://doi.org/10.1109/DEXA.2016.037
83. Vossen, G.: ACID Properties, pp. 1–3. Springer New York, New York, NY (2016). https://doi.org/10.1007/978-1-4899-7993-3_831-2. https://doi.org/10.1007/978-1-4899-7993-3%5F831-2
84. Wang, D., Rundensteiner, E.A., Ellison III, R.T.: Active complex event processing over event streams. Proc. VLDB Endow. **4**(10), 634–645 (2011). https://doi.org/10.14778/2021017.2021021
85. Winter, C., Schmidt, T., Neumann, T., Kemper, A.: Meet me halfway: split maintenance of continuous views. Proc. VLDB Endow. **13**(11), 2620–2633 (2020)
86. Wu, Y., Arulraj, J., Lin, J., Xian, R., Pavlo, A.: An empirical evaluation of in-memory multi-version concurrency control. Proc. VLDB Endow. **10**(7), 781–792 (2017). https://doi.org/10.14778/3067421.3067427
87. Zeuch, S., Chaudhary, A., Monte, B.D., Gavriilidis, H., Giouroukis, D., Grulich, P.M., Breß, S., Traub, J., Markl, V.: The nebulastream platform for data and application management in the internet of things. In: CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12–15, 2020, Online Proceedings. www.cidrdb.org (2020). http://cidrdb.org/cidr2020/papers/p7-zeuch-cidr20.pdf
88. Zhang, S., Wu, Y., Zhang, F., He, B.: Towards concurrent stateful stream processing on multicore processors. In: 2020 IEEE 36th International Conference on Data Engineering (ICDE), pp. 1537–1548 (2020). https://doi.org/10.1109/ICDE48307.2020.00136
89. Zhang, S., Zhang, F., Wu, Y., He, B., Johns, P.: Hardware-conscious stream processing: a survey. SIGMOD Rec. **48**(4), 18–29 (2020). https://doi.org/10.1145/3385658.3385662